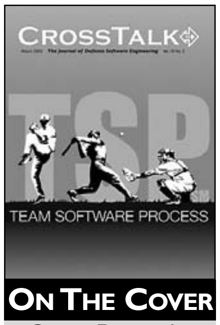
CrossTalk

March 2005 **The Journal of Defense Software Engineering** Vol. 18 No. 3

TEAM SOFTWARE PROCESS



On The Cover
Cover Design by Kent Bingham.

CrossTalk

OC-ALC/ MAS Co-Sponsor	*Kevin Stamey*
OO-ALC/MAS Co-Sponsor	*Randy Hill*
WR-ALC/MAS Co-Sponsor	*Tom Christian*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Palmer*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Creative Services Coordinator	*Janna Kay Jensen*
Phone	(801) 775-5555
Fax	(801) 777-8069
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/ crosstalk

Oklahoma City-Air Logistics Center (OC-ALC), Ogden-Air Logistics Center (OO-ALC), and Warner Robins-Air Logistics Center (WR-ALC) MAS Software Divisions are the official co-sponsors of CrossTalk, The Journal of Defense Software Engineering. The MAS Software Divisions and the Software Technology Support Center (STSC) are working jointly to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

The STSC is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal.

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 29.

OO ALC/MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-7026, or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

From the Sponsor

Team Software Process Brings Project Success Over Time

While Personal Software Process^{SM} (PSP^{SM}) has proven benefits, it has nothing to offer if employees are not using it. The Ogden-Air Logistics Center faced this problem in 1998 when developing the TaskView software, which allows the user to view quickly an Air Tasking Order from low-level detail to high-level overview. More than 20 individuals in our software engineering division (MAS) had received PSP training, but six months after the class only one person was using it. As a result, MAS and TaskView became a test site for the Software Engineering Institute's (SEI^{SM}) Team Software Process^{SM} (TSP^{SM}), and Watts Humphrey, a SEI Fellow who led development of the TSP and PSP, worked directly with TaskView on this SEI pilot project.

Over the next four years, the team internalized the TSP/PSP concepts and delivered several remarkably successful releases of the TaskView product. As instructed by Humphrey, the TaskView team organized, used the scripts and templates, held weekly meetings, and updated its spreadsheets. Each team member also employed PSP processes, checklists, and spreadsheets personally. They also developed a *dashboard* application to collect semi-automatically PSP data. As a result, they reaped the TSP/PSP rewards: low defects, greater flexibility, more accurate estimates, team unity, quality focus, delivered on-time and within budget, and delighted the customer.

Of course, this wasn't an airtight success. There were misunderstandings in recording data, some PSP training was incomplete, PSP tailoring was needed, and other issues just like any project. In other words, TSP/PSP wasn't a quick fix. It was, however, the key to creating a very disciplined and self-guiding team. The success of the TaskView team led to not only the adoption of TSP in other mission planning and Command, Control, Communications and Intelligence projects in MAS, but helped guide many high maturity policy and process improvements in the division that are still in use today.

Randy B. Hill

Randy B. Hill
Ogden Air Logistics Center, Co-Sponsor

From the Publisher

TSP Has Multiple Uses

The Team Software Process^{SM} (TSP^{SM}) has proven successful in helping software projects and organizations adequately plan their efforts, track against those plans, and take action in a timely manner when things don't go according to plan. I'm not just parroting the information I have read in this month's articles: I used TSP for two years before coming to work for CrossTalk. While I have seen its great advantages, I have also experienced its challenges. This month's issue contains articles that discuss both.

Alan S. Koch highlights the similarities between TSP practices and the practices advocated in the Capability Maturity Model® (CMM®) and CMM Integration^{SM} (CMMI®), and how implementing TSP can actually facilitate your CMMI process improvement effort. Carol A. Grojean gives several examples of how TSP improved her development efforts and provides data to back her claims. Ray Trechter and Iraj Hirmanpour share some of the challenges that can come with TSP success. David Webb and David Tuma explain in their article that TSP itself does not cause success – the practices it contains do: TSP has a framework that guides developers to these practices.

TSP is also evolving to meet the needs of new applications and processes. Chris A. Rickets' article discusses how his command adapted the published TSP process to meet the needs of its maintenance projects. In conclusion, Watts Humphrey presents an informative overview of why projects truly need TSP and insights into how it works. We received more good articles for this issue than we could possibly fit into it, so look for more TSP articles in later CrossTalk issues.

Elizabeth Starrett

Elizabeth Starrett
Associate Publisher

TSP Can Be the Building Blocks for CMMI

Alan S. Koch
ASK Process, Inc.

Your organization has a mandate to achieve Capability Maturity Model® Integration (CMMI®) Level 3. Why would you even consider adding the Team Software Process^{SM} *(TSP*^{SM}*) to your plate when it is already overflowing? In this article, I will discuss how TSP – far from adding work to a CMMI initiative – can potentially reduce the time and effort that will be required to achieve your goals. Simultaneously, TSP will engage your engineers in disciplined processes, giving them an appreciation for good processes along with the desire to adopt improved processes in every area of the organization.*

Your organization has a mandate to achieve Capability Maturity Model® (CMM®) Integration (CMMI®) Level 3. It is going to be a long road. There are processes to define, documents to write, people to train, and evidence to collect. With so much to do and so many people involved, why would you even consider the Team Software Process^{SM} (TSP^{SM})? Why add something else to your plate when it is already overflowing with work? Most organizations adopt the TSP to achieve significant improvements in product quality, to reduce development time, and to get more accurate project estimates. Too many organizations believe they must choose between those benefits and the recognized organizational maturity level or capability level rating provided by the CMMI.

In this article, I will discuss how the TSP can potentially reduce the time and effort required to achieve your CMMI goals, eliminating the need to choose between two sets of laudable objectives.

If you are just starting out on a CMMI initiative, the TSP can help you to bootstrap your process definition activities. TSP does this by providing starting points for many of your new processes, as well as training your engineers to be capable and productive at defining the processes they use. Naturally, the more progress you have already made in advancing the CMMI, the less this bootstrapping will help. But even if you are well on your way to your CMMI goals, the TSP can make the road smoother in these ways, for example:

- The Personal Software Process^{SM} (PSP^{SM}) training is a potent tool for overcoming the natural resistance that engineers often display toward process changes, and the TSP launch and weekly process can diffuse any remaining reservations they might have. By the time the TSP team is working together, most resistance has been worked through, so they become proponents of process change rather than resistors. Instead of having to *push* process improvements on these software engineers, you may find that they *pull* eagerly for them.
- Your TSP projects will yield cost savings, productivity improvements, and quality advances that will more than pay for the costs of introduction. In fact, even with good CMMI processes in place, your TSP projects will accelerate the rate of return on your entire process improvement initiative, allowing you to either accelerate your process work or realize the returns on it earlier.

® Capability Maturity Model, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
SM CMM Integration, Team Software Process, Personal Software Process, TSP, PSP, and SEI are service marks of Carnegie Mellon University.

Figure 1: *TSP Versus CMM [1]*

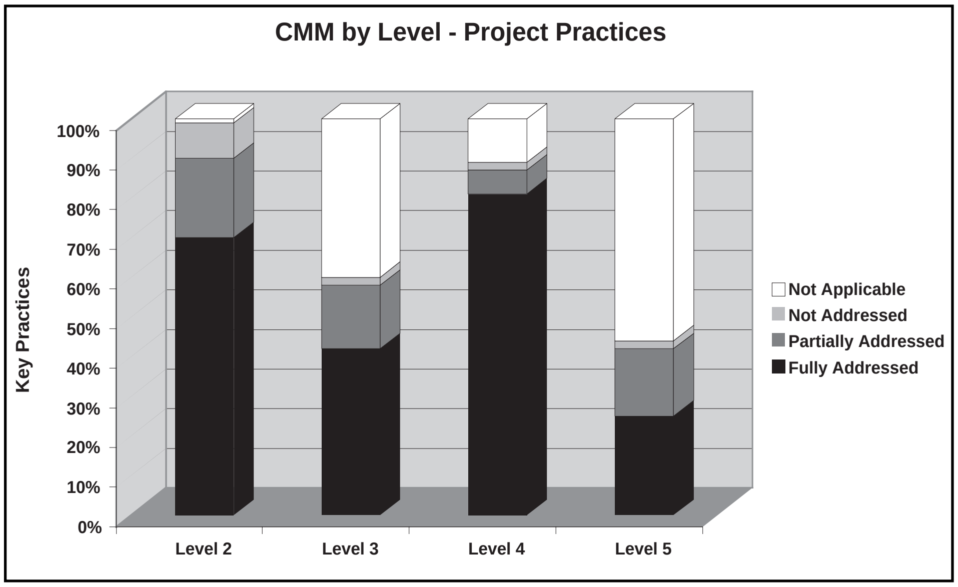


Figure Legend
Not Applicable: The content of the key practice does not apply in the scope of the TSP (key practices that deal with organizational activities are not applicable to the TSP).
Not Addressed: The TSP does not address any of the content of the key practice.
Partially Addressed: The TSP covers part of the content of the key practice.
Fully Addressed: The TSP fully covers the content of the key practice.

TSP and the CMMI

The TSP has the same roots as the CMMI, being based on the Software Engineering Institute's (SEI^{SM}) early research that produced the CMM for Software (SW-CMM). As such, it aligns well with the CMMI and partly or fully satisfies the CMMI's goals. Figure 1, which was published by the SEI in 2002, shows the degree to which the TSP addresses the key practices of the SW-CMM.

Figure 1 shows that the majority of the key practices in the CMM are at least partially addressed by the TSP. However, how are they addressed in the CMMI? The SEI is expected to publish a similar analysis of the TSP versus the CMMI soon, and because of the similarities between the models, we should expect similar results. In a presentation at the 2003 Software Engineering Process

Group conference [2], the SEI reported these preliminary findings about the specific practices (SP) and generic practices (GP) of the CMMI:

- **Project Management SPs:** Most fully or largely implemented.
- **Process Management SPs:** Majority partially or largely implemented.
- **Engineering SPs:** Majority fully or largely implemented.
- **Support SPs:** No consistent pattern as yet.
- **Generic Practices:** No policies in the TSP, but most other GPs at all capability levels are either taught in PSP training or practiced by TSP teams, or both.

What Is the PSP and TSP?

How do PSP and TSP address these CMMI goals? They do so through rigorous training followed by establishing a team environment that encourages the trained engineers to use in their day-to-day work what they learned in the training.

PSP

The first step in a TSP initiative is to train all team members who can write programs in PSP. The PSP is more than just training; it is a boot camp consisting of about 40 hours of classroom instruction, 10 programming assignments, and three data-analysis exercises, requiring a total of about 150 hours for the average programmer to complete.

The result of the PSP boot camp is that the programmers do not just learn *about* good processes, they actually improve their own processes, measure the effects of those process changes, quantify the benefits they have experienced, and set goals for further improvements. The PSP achieves these results by leading students through three steps.

1. In PSP0, they lay a simple foundation for the learning to come:
   - Following simple process scripts.
   - Collecting three basic measures of their work (time spent, size of products produced, and defects corrected).
   - Performing a simple post-project analysis.
2. In PSP1, they begin to build the capability to plan and manage their own work:
   - Following a defined project planning process.
   - Using their own prior data to make increasingly more accurate estimates for each programming assignment.
   - Planning their work at a level of detail that allows them to track and manage their progress.
3. In PSP2, they focus on achieving significant quality improvements:
   - Using their prior data to plan for incremental improvements in the quality of their programs.
   - Removing defects early using personal review techniques guided by their own prior defect performance.
   - Identifying and capitalizing on defect prevention opportunities in their program design and implementation methods.

Those who complete the PSP boot camp emerge with the knowledge and skills to make accurate plans, work to those plans, and produce superior quality products.

"The TSP has the same roots as the CMMI, being based on the Software Engineering Institute's early research that produced the CMM for Software. As such, it aligns well with the CMMI and partly or fully satisfies the CMMI's goals."

TSP

The TSP then provides the project framework in which programmers can carry these classroom skills back to the workplace and use them to transform their team's performance. Although the TSP is not primarily about training, it *does* include training for certain players:

- The *TSP Executive Strategy Session* (which is usually held before the engineers receive PSP training) provides senior managers with the opportunity to identify key software development-related issues, determine how the PSP/TSP will address those issues, and outline a strategy for making significant progress on those issues in a timely and cost-effective way.
- *Managing TSP Teams* training provides lower-level managers with the tools and methods for making the most of their TSP teams' ability to estimate accurately, plan appropriately, and self-manage.
- *Introduction to Personal Processes* allows the non-programmers on the TSP teams to learn about the same topics that the programmers learned in the PSP boot camp, though without the intensive work.

With all these pieces in place, the team is ready for the key activity of the TSP: the project launch. This is a four-day workshop in which the TSP team members estimate and plan their project, utilizing goals, objectives, requirements and constraints from senior management and clients, along with their data on their own performance on prior projects. The launch culminates in a presentation by the TSP team of the project plans to senior management and client, and, often, negotiation of goals and requirements in light of constraints and expected performance. The final result is a project plan that is aggressive yet achievable, and is agreed to by all stakeholders.

After the launch, the TSP also provides the team with processes and tools for regular progress monitoring, identifying and acting on corrective actions, and reporting status up the management chain. For projects that take longer than a few months, TSP relaunches provide a basis for incremental project planning and regular realignment of plans with project progress.

PSP and TSP Costs

As can be seen from the descriptions of PSP and TSP, the largest cost component of introducing TSP in an organization is the time the engineers spend in training and project launches. Each engineer spends three to four weeks in PSP training (usually spread over two to three months), and all project participants spend nearly a week in each project launch.

These costs are embraced with the expectation that the time invested will be returned at the end of the first project as the system testing time shrinks due to the improved quality of the system produced by the TSP-trained team.

TSP Need Not Be Additional Work

How can I say that the *TSP need not be additional work* when I just discussed the amount of time and effort required to introduce it? That is work that an organization certainly would not be engaged in if it did the CMMI alone! This is true. But the point of this discussion is that by investing in the TSP, you can make substantial *reductions* in the time and effort for the overall CMMI effort.

This is suggested in reports published in CROSSTALK about CMM implementation at the AV-8B Joint System Support Activity (JSSA), at China Lake, Calif. In September 2002 [3], JSSA reported achieving CMM Level 2 in a relatively quick 14 months by adding the TSP to its strategy. Then, in January 2004, JSSA made a follow-up report about moving from CMM Level 2 to Level 4 in only 16 months (as opposed to the normal 50 months), crediting the TSP with their almost unheard-of pace. JSSA's experience shows that you can capitalize on the TSP's proven framework for process improvement to speed your CMMI initiative along.

How can this be? As discussed in the earlier analysis, the TSP addresses many of the same subjects as the CMMI, but it does so from the opposite perspective. Where the CMMI takes an organizational perspective, the TSP comes at these topics from the perspective of the individual engineers and the teams in which they operate. This perspective has two main advantages.

First, the TSP builds understanding and acceptance of process discipline from the grass roots up. The PSP (as the first step of a TSP initiative) builds understanding of the important principles in the individual engineers. The PSP boot camp for engineers not only teaches these important principles, but it provides the setting in which each engineer can try them out and prove their worth based on personal experience and data. The TSP then shows the engineers how to apply those same principles on real projects in a team setting.

The second main advantage is that the TSP works in the small. Anyone who has been involved in organizational change knows of the difficulty in changing the way that people work. Even if they are not hostile to the change, most people will be reticent to abandon the tried-and-true for the unknown. By focusing on one team at a time (and teams of PSP-trained individuals at that), the TSP mitigates many of the difficulties inherent in organizational change.

Taken together, these two advantages give the TSP the potential to accelerate the organization's process improvement effort not by initiating yet another effort, but by including the TSP as a strategic part of the larger CMMI improvement initiative.

Get Engineers Involved at the Beginning

The CMMI by its nature tends to not affect many of the engineers in the early stages of an improvement effort. This is because the first process areas (PA) the organization works on at Level 2 are generally focused on management rather than engineering. The Project Planning, Project Monitoring and Control, Supplier Agreement Management, and Measurement and Analysis PAs tend to deal with topics that only marginally affect engineers. Even the Requirements Management PA (as opposed to the Requirements Development PA) is more about management issues than technical ones.

"The metrics practices the engineers learn in the PSP and TSP have the potential to form the basis for the organization's MA [measurement and analysis] processes and procedures."

The unfortunate result of this management focus at Level 2 is that the engineers tend not to be engaged in the effort, and either feel that they are being left out or that they are lucky to avoid involvement. Of course, this situation changes dramatically when the organization begins focusing on Level 3 and its engineering-specific PAs such as Requirements Development, Technical Solution, Product Integration, Verification, and Validation.

By making the TSP an integral part of your CMMI improvement effort, you assure your software engineers are facile with topics that are traditionally the territory of managers, and the focus at CMMI Level 2 (measurement and analysis, project planning, and project monitoring and control), while building grassroots acceptance of process discipline and establishing good processes in each team.

In the following three sections, I will look specifically at the PAs that are the primary focus of the TSP. They are measurement and analysis (MA), project planning (PP), project monitoring and control (PMC) at Level 2, and verification (Ver) at Level 3. Because of the TSP's primary focus on these PAs, TSP teams can play a critical role in working out and pilot-testing processes for them.

Measurement and Analysis

PSP training starts out by showing the engineer how to collect three primary measures: time, size, and defects. It focuses on these three (as the engineers soon learn) because they are critical to achieving the PSP's goals of accurate planning and quality management. As the engineers go through the training, they are encouraged to analyze their individual data to understand how it illuminates their performance, and to use it as the basis for any process improvements.

The TSP builds on this basic understanding of a few measures by prompting each team to identify the metrics they will need based on the goals of the project, then to collect and analyze those metrics regularly. So the TSP implements MA at the individual project level, starting with goals and objectives, identifying metrics to support them, and collecting and using them throughout the project.

The metrics practices the engineers learn in the PSP and TSP have the potential to form the basis for the organization's MA [measurement and analysis] processes and procedures. In addition, as these are developed, TSP teams will provide a perfect infrastructure in which to pilot test them. Since the TSP team members will understand metrics and how to collect and use them, they will provide the feedback you require to evaluate the effectiveness of those candidate processes and procedures.

If you have already established your organization's MA processes and procedures, then your TSP teams will use them just as any other project team does. The only difference is likely to be the enthusiasm with which they embrace metrics and the constructive criticism they will provide to help you with improving your MA procedures and standards.

Project Planning and Control

After instituting the collection of basic metrics, the PSP teaches engineers to use their own data to plan their individual work. They analyze their assignments in light of the projects they have undertaken to date, and use their own data to make reasoned and achievable plans for completing it. They then learn to compare their actual performance against those plans so they can improve their future plans, in addition to seeking ways to improve their processes.

The TSP harnesses these skills to lead the engineers through estimating and planning the work for the entire team. After each team member is assigned specific work items, he or she uses personal data to produce a detailed individual work plan, and those plans are rolled together and balanced to produce the team's final project plan. Then every week, the team collects and analyzes the data on its actual performance and if deviations from their plan call for it, they identify, implement, and track corrective actions.

The practices your engineers will learn in the PSP and TSP can form a solid foundation for your organization's PP and PMC processes and procedures. Your engineers will be able to tell you how well those practices work in your organization as well as providing ideas for ways to fine-tune them to their specific needs. Then, as you document your organizational PP and PMC processes and procedures, they will easily be able to incorporate them into their ways of working and give you feedback on how well they work. With the PSP and TSP as a foundation, establishing your organization's PP and PMC processes should be relatively easy and fast.

If you have already established your organization's PP and PMC processes and procedures, then your TSP-trained engineers will use them as the basis for their own planning and tracking activities. But because of the team's deepening understanding of the mechanisms for planning and tracking projects, they will be a regular source of suggestions for improving your organizational PP and PMC standards to better fit the organization's needs.

Verification

The other major focus of the PSP and TSP is on quality management. The PSP teaches the engineers how to track their defects, and then use that information along with good review processes to do an effective job of personal design and code reviews. The TSP adds peer reviews, giving the team members the tools and methods they need to remove the vast majority of defects from their programs *before* they begin testing, thus improving product quality *and* reducing system test time.

All of the basic processes and procedures that you will need for the Ver PA are embodied in the TSP's review methods. And, as with the PAs discussed so far, your TSP teams will be ready to pilot test your organization's Ver processes and procedures after they have been defined. And again, if you have already defined your organizational Ver processes and procedures, then your TSP teams will embrace them and provide improvement ideas for them.

Defining Your Other Processes

In addition to the four PAs discussed, your TSP teams will be indispensable as you develop the processes and procedures for all of the other PAs. Because the PSP and TSP address most of the CMMI PAs to at least *some* extent, your TSP teams will have at least some experience with most of the PAs that you will need to address.

As you begin working on any particular PA, your first step should be to discuss it with your TSP team members. As you find out how they address that particular PA, you will find that at least some of the procedures you will need already exist. Then, as you explore more deeply to expand those processes to fully satisfy the goals of the CMMI, your TSP team members will be likely to have good ideas about how to add any missing practices, or how to tune those that in some way fall short.

Because of their experience with using and improving disciplined processes, they will be strong members of any process action team that you establish. They will contribute practical ideas to meet the CMMI goals, and they will be able to evaluate alternatives from the basis of practical experience. TSP team members will help to make your process action teams effective at defining and documenting processes and procedures that will work in your organization.

Summary

Adding the TSP to a CMMI initiative does *not* mean adding more work to an already overworked group. Rather, it can be an effective way to accelerate that initiative by laying a solid foundation of process discipline, engaging the engineers from the very beginning, and providing processes that already address a significant number of CMMI practices. The costs of incorporating the TSP into your CMMI initiative should be more than returned as you achieve your CMMI goals more quickly and with less organizational pain.

In addition, by adopting the TSP, you can convert your engineers into allies in the process improvement initiative – people who will lobby for better processes, help you to realize them, and embrace them in their day-to-day work.◆

References

1. Davis, Noopur, and Jim McHale. "Relating the Team Software Process^{SM} (TSP^{SM}) to the Capability Maturity Model® for Software (SW-CMM®)." Pittsburgh, PA: Software Engineering Institute, Mar. 2003 <ftp://ftp.sei.cmu.edu/pub/documents/02.reports/pdf/02tr008.pdf>.
2. McHale, Jim. "The Case for Using TSP^{SM} With CMM®/CMMI®." Software Engineering Process Group (SEPG) 2003, Boston, MA <www.sei.cmu.edu/cmmi/presentations/sepg03.presentations/tsp.pdf>.
3. Hefley, Bill, Jeff Schwalb, and Lisa Pracchia. "AV-8B's Experience Using the TSP to Accelerate SW-CMM Adoption." CROSSTALK Sept. 2002 <www.stsc.hill.af.mil/crosstalk/2002/09/hefley.html>.

Additional Reading

1. Pracchia, Lisa. "The AV-8B Team Learns Synergy of EVM and TSP Accelerates Software Process Improvement." CROSSTALK Jan. 2004 <www.stsc.hill.af.mil/crosstalk/2004/01/0401pracchia.html>.

About the Author

Alan S. Koch is president of ASK Process, Inc., a training and consulting company that helps organizations to improve the return on their software investment by focusing on the quality of their software products and the processes they use to develop them. Koch was with the Software Engineering Institute (SEI^{SM}) for 13 years where he became familiar with the Capability Maturity Model®, earned the authorization to teach the Personal Software Process^{SM} (PSP^{SM}), and worked with Watts Humphrey in pilot testing the Team Software Process^{SM} (TSP^{SM}). He also worked in software development for 28 years. Koch is an SEI-authorized PSP Instructor and TSP Coach candidate, an SEI Transition Partner for the PSP and TSP, and a certified Project Management Professional. He is a member of the Project Management Institute and author of "Agile Software Development: Evaluating the Methods for your Organization."

ASK Process, Inc.
4378 Ridge RD
Natrona Heights, PA 15066
Phone: (412) 849-0421
E-mail: alan.s.koch
@askprocess.com

Microsoft's IT Organization Uses PSP/TSP to Achieve Engineering Excellence

Carol A. Grojean
Microsoft Corporation

Projects today are beset with problems from the very beginning. Many of these problems come from outside the team, be it over-ambitious deadlines from management, last-minute scope changes from the customer, or not enough resources. But not all problems can be blamed on these issues; many of the projects' problems come from within: failure to plan, failure to track actual progress against the original plan, failure to include changes in the plan, poor estimation, and failure to understand when effort deviates from the plan. The Personal Software Process^{SM} teaches engineers that estimating and planning are a big part of their job, and that what they predict drives all other efforts. At the Team Software Process^{SM} level, we bring the team together to define their processes and make a detailed plan. The process then requires the team to enforce their commitment as well as their individual behavior to follow the process to track time, use data, and get high quality.

The success of organizations that produce software-intensive systems depends on well-managed software development processes. Implementing disciplined software methods, however, is often challenging. Organizations seem to know what they *want* their teams to be doing, but they struggle with *how* to do it [1].

Unfortunately, when consistently challenged with a schedule, the defect elimination process continually gets pushed later and later into the development cycle and, eventually, over-the-wall to test so the engineers can continue to be *productive* and meet their schedule-driven goals. Many organizations fail to truly understand the impact poor quality has on their ability to meet schedule commitments.

Peter Russo, general manager for Microsoft's information technology (IT) application architecture group comments that:

> There are two fundamental issues in most IT organizations today, one being the ability to accurately predict a project schedule, and the other being the quality of the product once you are finally done – and these are two challenges we have to start addressing today.

Data Analysis and Findings

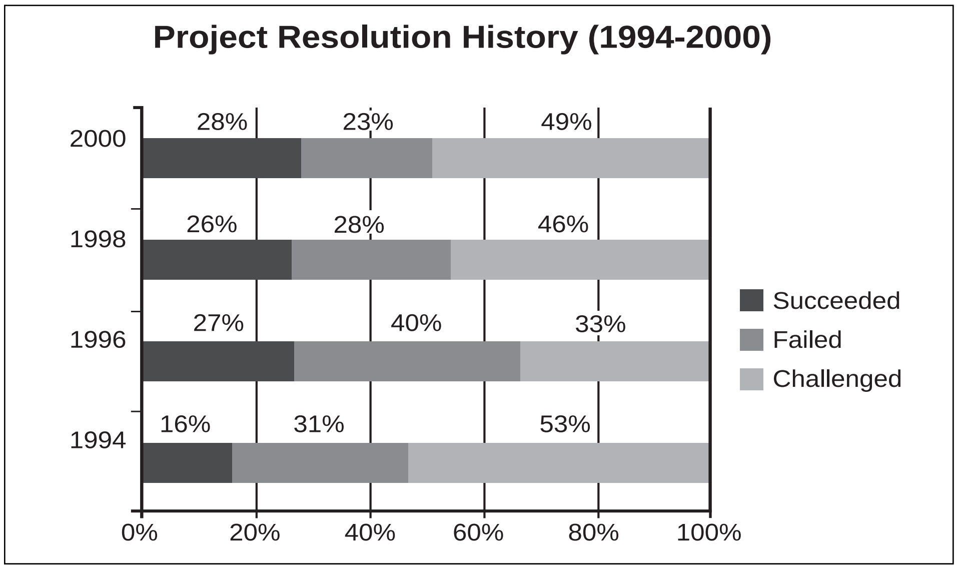
Most software organizations are facing critical business needs for better cost and schedule management, effective quality management, and cycle-time reduction [2].

In 1994, the Standish Group reported in their famous "Chaos Report" [3] that only 16 percent of projects succeed while 31 percent fail and 53 percent are significantly challenged, with the average project running approximately 189 percent over schedule (see Figure 1). In 2000, while things appear to be getting a little better, we still have a ways to go.

Furthermore, the Standish Group cites that while numbers appear better, that is not the entire story: many projects are overly estimated.

> In a number of focus groups, IT executives told us that they first get their best estimate, multiply by two and then add a half! It should not be surprising, therefore, that the majority of these successful projects were already 150 percent over budget before they began! [3]

Figure 1: *Project Resolution History [3]*



Most organizations have a schedule and a fairly detailed plan, and generally their plan goes one of two ways: either they have a development cycle followed by a test cycle where they expect the engineers to just fix bugs during the test cycle, or their plan has the engineers move on to developing the next set of features, setting aside a little time in case any bugs come up. In either case, the amount of time the engineer needs to spend regressing and fixing defects found by test is woefully underestimated every time. Additionally, once a product is released to the customer, management assumes the engineer is free to move on to the next project or cycle of the product. The plans generally do not consider that defects from the first release will take engineers away from their progress on the second release – this is how the vicious cycle begins.

The gap of quality code means that engineers are spending too much time fixing bugs, either from the previous release or the current release, on code already passed along to test. In doing so, they cannot make progress on new work as originally planned. This conflict creates a tension in the cycle where there needs to be a balance between injecting defects and removing them. The current process of engineer-injecting and test-removing is not a natural balance: The system pushes back by surfacing all the defects that testing cannot find to later in the cycle. The only solution is to understand that balance has to exist within the development cycle, which Personal Software Process^{SM} (PSP^{SM})/Team Software Process^{SM} (TSP^{SM}) helps us understand.

Life at Microsoft

As it turns out, life in Microsoft's IT organization is not a whole lot different than many other IT departments when it comes to development. Many IT managers face decreased headcount and budgets with increasing support costs, as well as projects with unpredictable schedules and lower quality than they would like to see. As IT manager Todd Baumeister puts it:

> Today's projects are estimated and managed not with data, but on the gut instinct of the developer – and with this I have to go to the customer with conviction to our project schedule and then refute their asks of wanting more in less time, and this isn't a position I want to be in … I want a predictable schedule that will demonstrate our ability to plan for a date and hit that date and when we deliver the product it will be of high quality.

Every other IT manager I interviewed had a similar story to tell, and many added that they would like to see their team's morale increase. Another IT manager stated that he would like to basically "deliver a high quality product on a predictable schedule (when we said we would) without any death marches or dead bodies left behind."

Fundamental to all was the need to get developers focusing on features and design and delivering a high-quality product so that test can focus on performance, reliability, security, and ensuring the customer's needs are met, not this finding-fixing-testing loop that exists today.

About two and a half years ago, Microsoft IT Manager Aidan Waine was on a plane from Seattle to Reno reading Watts Humphrey's book "Winning with Software." Waine's development projects were out of control with high bug counts, ever-increasing test cycles and low delivery predictability. Client satisfaction and engineering team morale were both low. This book directly addressed these issues, describing controlled high-quality software delivery through disciplined, repeatable, data-driven engineering processes. Waine bought another 28 copies for his management team and clients. He jokes that no one read the book or took him seriously, so he went one step further and brought Humphrey out to Microsoft. Senior management bought into the experiment – two development teams were trained in the new methods, and two TSP projects launched.

Results so far have ranged from good to amazing. Waine said that the very first project he ran in his newly organized team more than paid for the training. His initial reaction was, "Wow, we're giving them two weeks of training and look what we got back!" The result of their first project, though small, was all they needed.

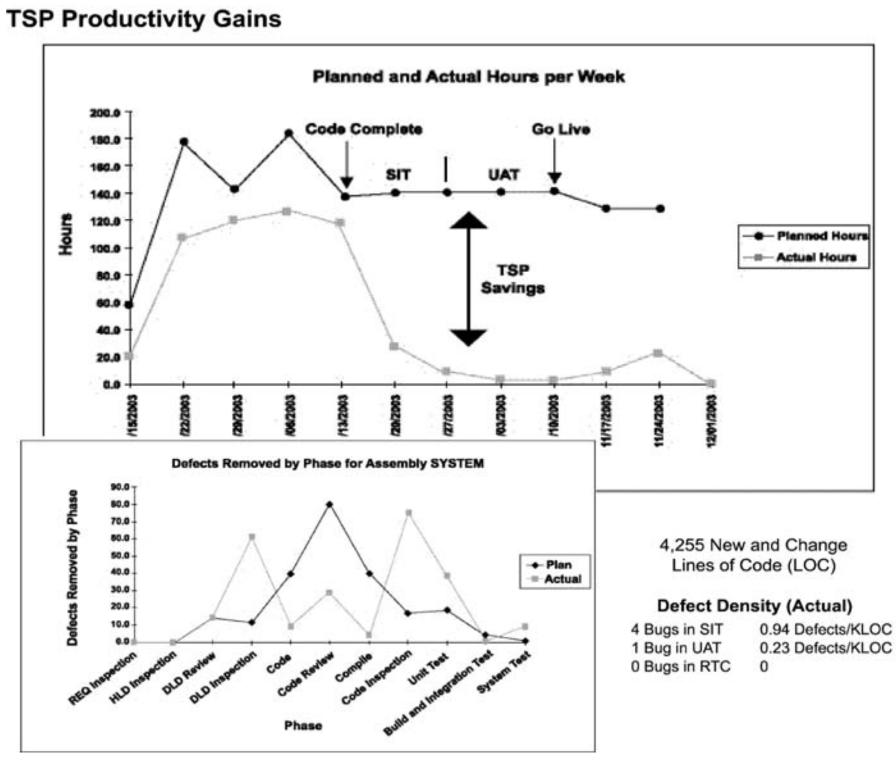


Figure 2: *TSP Productivity Gains*

As you see in Figure 2, for 4,255 new and changed lines of code, Waine's team had only five defects from the time the product went to system integration test (SIT) to user acceptance test (UAT) to released to the customer (RTC). The engineers were bored in test – can you imagine that?

When asked what challenges they faced once they made the decision to pilot or deploy PSP/TSP, all the managers interviewed were senior enough that they understood that they needed to lead by example, and that executive support and involvement is the No. 1 driver in project success. They understood that what they were asking for was a change of behavior from the individuals as well as something that goes against the culture of the organization, not to mention the software industry itself: collecting your data around time, size, and defects.

For the most part, they also mandated the change. This understanding and executive sponsorship sent a common and consistent message that change was expected and as a result, they faced very little political pushback. The biggest administrative challenge they faced was finding the time to get the engineers trained. "People don't come off the shelf PSP trained," one manager exclaimed. However, they understood that this had to happen, and that no time was going to be a good time. Moreover, they found it increasingly hard to find good, available coaches to lead their teams.

Additionally, there was the management challenge of convincing the business to take a four-week hit (for training and launching) with some sort of promissory note that it would be worth it in the end – a story they had been sold before. Fortunately, as more and more projects were finishing up, customers began to hear success stories from their counterparts and asked for the same results on their IT-driven projects.

"I'm excited about this," says Microsoft Chief Information Officer Ron Markezich, referencing a recent Accenture study titled "Value Discovery: A Better Way to IT Investments," a survey of 100 large European IT shops that showed astonishing results [4]. Up to 60 percent to 70 percent of most of an IT shop's budget is spent on sustainer activities such as support and fixing bugs. Markezich said:

> I'm excited because the potential for this to not only reduce our product cycles and increase our quality, but ultimately freeing up much of our sustainer activities enables us to invest more in builder activities that drive more value to the business.

What Do the Engineers Think?

A little over a year ago, I had the opportunity to launch an internal tools team that was literally broken down, having just come off a project that was cancelled after two and a half years of effort. According to Software Engineer Vivek Rao:

> I had just started at Microsoft when our team decided to adopt TSP/PSP. I did not know then, but would learn later, that the team was lagging in their interface with customers and was not listening to their needs. The team did the two-week PSP training of which I was a part. In the beginning, I was very skeptical of the idea, since it seemed like too much process to me. I felt it would stifle innovation. However, during the training I saw the importance of spending time to review my design and code. The satisfaction that I was getting by having zero defects in compile and an essentially defect-free program was invaluable. I also realized that I was a lot more confident about the code than I ever was. There was clearly an idea of design, code, and review both and be done with it, rather than keep fixing bugs later on, forever. I quickly realized the contribution of the PSP process toward quality of the code.

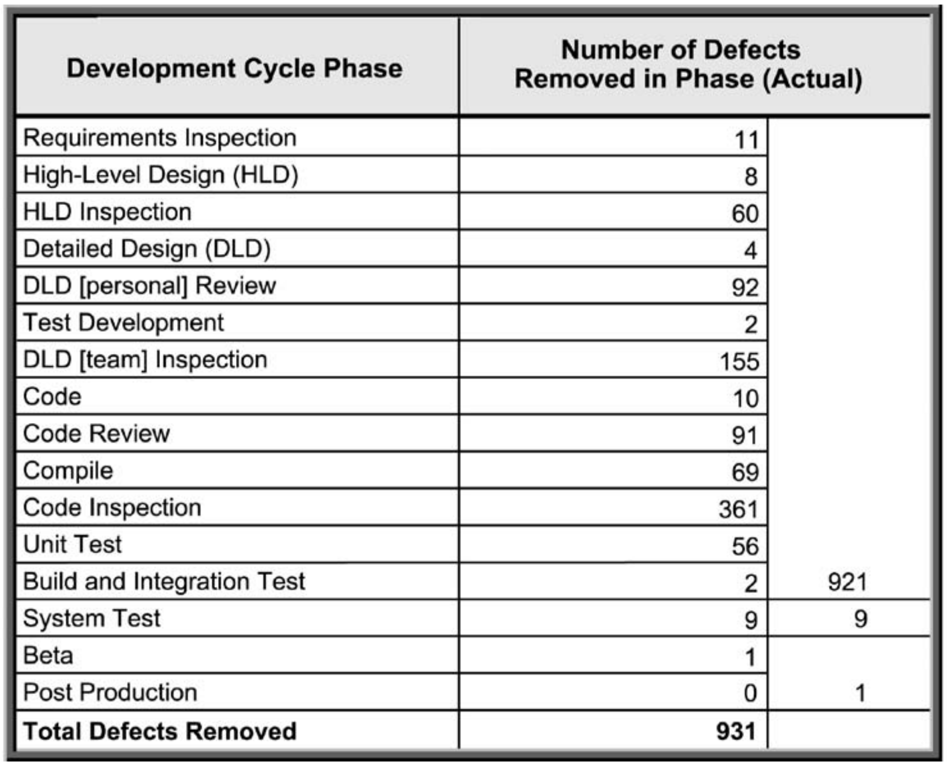
"It is very hard to convince people to change until they have to," said Watts Humphrey, "and the power of the PSP is that they see things change effectively" [5]. With PSP, the engineers honestly begin to see the change and begin to understand that to do good work you have to be disciplined, but change is hard. "People want to fight change and fight the notion of tracking time, size, and defect information," Humphrey said. "Most engineers just want to write code and don't perceive the other phases of the project (planning, design, reviews and inspections, etc.) as value-added, though they generally know they are good things to do."

That is the good part about PSP/TSP: earned value helps measure time all the way through the product development cycles, and people realize the importance of planning before design, design before coding, code reviews and inspections, etc. They also start seeing little changes right away – a code compiles without defects (or many fewer than before) or a test is virtually defect-free, which is something they never believed possible before. When the product gets to test and they are not scrambling around fixing bugs, they quickly become believers. Furthermore, they do not spend a lot of time arguing severity or priority of a bug because they have time to fix everything.

Vivek stresses this and goes on to say:

> The team then went into launching the project, what TSP calls a launch. The amount of energy that was generated during the launch was amazing – it really gelled the team together. The launch made the customer requirements clear and helped the team obtain buy-off on the schedule from the customers. It also helped me see the big picture of the project and my dependencies. During the launch, we also modified some of the processes to fit our needs, and it was clear that TSP could be modified to suit a team's needs. We have made small changes to the process over time to make it more efficient for us, while at the same time ensuring quality.
>
> As the project proceeded, I quickly realized one additional benefit of TSP: as a newcomer to the team, I was not familiar with the code base. I started taking part in many design and code reviews and because of this, I learned a lot about the code from experienced team members. For a new hire, TSP is an excellent means to learn about the design and code of a product in a short period of time. TSP also generated in the team a new sense of ownership and commitment toward the customer needs. Although this is not a direct result of TSP, it has resulted in the team scoring a 10/10 in customer relations.
>
> To summarize, the proper application of PSP leads to an immense sense of achievement and satisfaction, and TSP furthers this to the team level, ultimately resulting in good products.

Table 1: *Defect Removal Profile by Phase*



Development Cycle Phase	Number of Defects Removed in Phase (Actual)	
Requirements Inspection	11	
High-Level Design (HLD)	8	
HLD Inspection	60	
Detailed Design (DLD)	4	
DLD [personal] Review	92	
Test Development	2	
DLD [team] Inspection	155	
Code	10	
Code Review	91	
Compile	69	
Code Inspection	361	
Unit Test	56	
Build and Integration Test	2	921
System Test	9	9
Beta	1	
Post Production	0	1
Total Defects Removed	**931**	

And the results of their first pilot speak volumes:

- Ninety-six percent schedule accuracy – finished two weeks late with three weeks of added features.
- Delivered 1.36 defects/thousand lines of code (KLOC) to system test.
- Huge improvement in partner satisfaction.

Specifically, their defect removal profile on 12,253 new and modified LOC looked like Table 1.

On 12,253 new and modified LOC, the team spent approximately 584.6 hours in review and inspection phases of the project (otherwise known as appraisal cost of quality), and spent 109.1 hours total for compile, system test, and user acceptance test phase (otherwise known as failure cost of quality) for a total of 693.7 hours finding and removing defects. This represented an appraisal to failure ratio of 6.36[1].

Prior to system test, they had removed 921 defects – a yield of 98.6 percent (meaning 98.6 percent of the defects injected into

Phase	Number of Defects Into Phase	Yield	Defects To Be Fixed in Phase	Average Time to Fix (each bug)	Total Fix Time	Actual Number of Defects	Actual Time Spent by Team
System Test	460	50%	230	4 hours	920 hours	9	28 hours
User/Beta Test	230	50%	115	8 hours	920 hours	1	6 hours
Release to Customer	115	50%	58	12 hours	696 hours	0	0 hours
Total					**2,536 Hours**	***versus***	**34 Hours**

Table 2: *Comparison of Typical Results*

the work product prior to system test were removed prior to system test). If the industry average were to be about 50 percent (many feel it is not that good, but I will be conservative) and had this team been average, then approximately 460 of the 921 defects would have slipped to system test or later phases. If system test had a 50 percent yield, then they would have had to remove at least 230 defects in system test. Time wise, most teams I coach plan for defects in system test to take about half a day to find and fix (on average), meaning they would have spent 920 hours finding and fixing defects in system test instead of the 28 hours that they actually spent.

Additionally, another 230 defects would have slipped to user/beta testing where, with another 50 percent yield and a cost of about a day, or eight hours, to find a fix, you have to find and fix an additional 115 defects at the cost of eight hours per defect (on average) to find and fix or a total of another 920 hours. Instead, they had no defects.

That leaves the product going to the customer with 115 defects which, on average, 50 percent of those will be found over the product lifetime (or a total of 696 hours to find and fix). That is a difference of 2,536 hours of finding and fixing bugs in our old way of doing things versus 34 hours they actually spent (a project savings of 2,502 hours). That is the difference of spending three-quarters of a week fixing bugs post-development versus 5.76 weeks for 11 engineers. This is illustrated in Table 2.

What Does the Customer Think?

If your IT organization is anything like ours, then you are probably continually pushing for change – whether it is with tools or the latest methodology or a new definition of your project life cycle. You are constantly striving to figure out how to get more out of your people for less.

I can only imagine what our internal customers thought when we approached them with this new process.

Cyndee Kraiger has been in Microsoft Operations for more than 11 years, and has been the recipient of many IT projects in that time. About 18 months ago, she pushed for a new tool to manage the operations for our volume license deliverables. This tool was to replace an extensive set of spreadsheets that had become so unmanageable that even a small mistake could cost Microsoft hundreds of thousands of dollars. While this project was the second TSP project for the Business Unit IT organization Kraiger was in, it was the first for her and she did not really know what to expect.

Kraiger had been through several projects before as the business owner. She said that this project had a different level of engagement than traditional projects from the very beginning. "More of my time was required but the content of the meetings were of high quality." She said that she was even given examples of what to expect from the project up front. "My team was engaged daily and felt very involved and committed at all times."

Other projects, she said, typically start out with a meeting in the beginning of the project with some sort of expectation being set (time, features, cost) and then another meeting in the end with what was developed. Of course, compromises happened along the way, generally without her knowing it. But with this project, her expectations were managed all the way through and at the end of the project, she was putting the bow on the wrapping, (not her typical experience). There were no surprises. The final result is in Table 3. The project was delivered on schedule. As you see, "I got what I wanted when it was promised and the product was of high quality," said Kraiger.

Summary

It would not be fair to blame all software problems on software developers. When we are consistently challenged with a schedule, the process of eliminating defects continually gets pushed back later and later into the cycle and, eventually, over-the-wall to test so that the engineers can continue to be *productive* and meet their schedule-driven goals; this is classic *shifting the burden* [6].

In this environment, we have our quick fix of giving the code to the test team to be fixed. Then we have the unintended consequence of defects coming back to us later to eventually fix, at the expense of the current project, which then falls behind. However, the pressure to continue to meet schedule milestones continues, and so the behavior continues as a project's schedule begins to atrophy in its ability to meet established schedules, quality statements, and/or features.

It is just too expensive for any organization to try to test quality in; it cannot be done. And without being able to accurately predict our project schedules and resource needs, we just cannot run our organization.

Jon DeVaan, senior vice president of Engineering Strategy at Microsoft frequently references the article "Nobody Ever Gets Credit for Fixing Problems That Never Happened" [7]. The article addresses the reality every manager faces: dedicating additional effort to either work or improvement can increase the performance of any process. The issue at hand is do you go down the destructive work-harder loop, where you feel short-term gains despite long-term consequences, or do you follow the constructive work-smarter loop, where you feel short-term pain for long-term investment in capability?

What strikes DeVaan most about this article [7] is the story about the BP team that reduced butane flare-off to zero in just two weeks, saving $1.5 million per year at a cost of about $5,000 to implement, creating a return on investment of 30,000 percent per year. The article reported that members of the team had known about the problem and how to solve it for eight years. They already had all the engineering know-how they needed and most of the equipment and materials were already on site. What had stopped them from solving the problem long ago? The only barrier was the mental model

Table 3: *Illustration of Final Results (Quality)*

New Lines of Codes	59,616	
Number of defects found in System Test (ST)	42	(0.705 defect density)
Number of defects found in User/Beta Acceptance Test (UAT)	5	(0.084 defect density)
Number of defects found in Production (to date)	9	(0.151 defect density)

Coming Events

April 4-6
Defense Technical Information Center (DTIC) Conference
Alexandria, VA
www.dtic.mil/dtic/annualconf/

April 5-7
Federal Office Systems Exposition (FOSE) 2005
Washington, DC
www.fose.com

April 18-21
2005 Systems and Software Technology Conference

Salt Lake City, UT
www.stc-online.org

May 2-6
Practical Software Quality and Testing (PSQT) 2005
Las Vegas, NV
www.qualityconferences.com

May 14-15
ACM Symposium on Software Visualization

St. Louis, MO
www.softvis.org/softvis05

May 15-21
27th International Conference on Software Engineering (ICSE)
St. Louis, MO
www.icse-conferences.org/2005

May 16-20
STAREAST 2005
Orlando, FL
www.sqe.com/stareast

May 23-26
2005 Combat Identification Systems Conference
Portsmouth, VA
www.usasymposium.com/combatid

(thinking) that there were no resources or time for improvement, that these problems were outside their control, and that they could never make a difference.

DeVaan emphasized that people should have the courage to change:

> Generally, most people know what the problem is and perhaps even how to fix it; the difficult part is just getting people to change. Everyone recognizes the problem and oftentimes it gets expressed over and over again in cynicism. The true insight is getting every level of management to understand that they are part of the problem when they continually reinforce the *work-harder* loop.

DeVaan further expressed that it takes a lot more guts to change the lower in the management chain you are. "At some point there has to be a line drawn where any management above the line is to the point of negligence for letting the behavior continue." He said that we need people to have the courage to take the heat when the drop (in productivity) is down, whether it comes from the board of directors, the chief executive officer, or the line manager.

We all have to get on the *work-smarter* track and recognize that long-term gains in process improvement do not come overnight – just as they were not created overnight.◆

References

1. Humphrey, W.S. "A Discipline for Software Engineering." 2nd ed. Manuscript submitted for publication. 2004.
2. Humphrey, W.S. Winning With Software: How to Transform Your Software Group Into a Competitive Asset. Boston: Addison-Wesley (Pearson Education), 2002.
3. Standish Group International, Inc. The Chaos Report. Standish Group International, Inc. <www.standishgroup.com/sample_research/PDFpages/chaos1994.pdf>.
4. Curtis, G.A., R. Melnicoff, and Tor Mesoy. "Value Discovery: A Better Way to IT Investments." Outlook 2003, No. 3 <www.accenture.com/xdoc/en/ideas/outlook/3_2003/pdf/info_technology.pdf>.
5. Humphrey, W.S. Personal interview. Aug. 2000.
6. Senge, P., et al. The Fifth Discipline: Strategies and Tools for Building a Learning Organization. New York City, NY: Doubleday, 1994.
7. Repenning, N., and J. Sterman. "Nobody Ever Gets Credit for Fixing Problems that Never Happened: Creating and Sustaining Process Improvement." California Management Review 4 (2001): 64-88 <http://search.epnet.com/direct.asp?an=5244741&db=bch&loginpage=Login.asp&site=ehost>.

Note

1. The appraisal-to-failure ratio is a measure of the cost of quality. Specifically, you want to measure the percentage of time you spend in appraisal phases of your cycle (such as design and code reviews and inspections) versus how much time you spend in failure phases (such as compile, system test, and customer test). Appraisal cost of quality (COQ) percentage is calculated as 100*(appraisal time)/(total development time). The failure COQ percentage is calculated by taking 100*(failure time)/(total development time). The total appraisal to failure ratio is then calculated by taking the percent appraisal COQ divided by percent failure COQ (percent appraisal COQ)/(percent failure COQ).

About the Author

Carol A. Grojean is a certified Personal Software Process℠ (PSP℠) instructor and Team Software Process℠ (TSP℠) Launch Coach and has been practicing TSP at Microsoft since May 2002 when she was the team leader of the company's first pilot project. Grojean was in Microsoft's IT organization for eight years before moving in to her current role as a member of the company's Quality Engineering Practices organization, helping to drive engineering best practices throughout the product group, including piloting PSP/TSP. Grojean has a Masters of Business in management information systems and a Masters of Science in project management and is a certified Project Management Professional.

Microsoft Corporation
One Microsoft WY
28/1230
Redmond, WA 98040
Phone: (425) 706-8903
Fax: (425) 706-7329
E-mail: cscott@microsoft.com

Experiences With the TSP Technology Insertion

Ray Trechter
Sandia National Laboratories

Iraj Hirmanpour
AMS, Inc.

Transitioning the Team Software Process^{SM} (TSP^{SM}) methodology into an organization is not an easy or simple task. It requires significant behavior change by not only the developers, but also the major stakeholders. In this article, we share three years of experience with the TSP transition team at Sandia National Laboratories with two intertwined perspectives: that of the TSP coach, and that of the development manager supervising the TSP projects.

The mission of the Information Systems Development Center of Sandia National Laboratories in Albuquerque, N.M. is to provide software development and support services for a variety of internal customers in support of their national defense and energy-related work. The organization's software development process, called Software and Information Life Cycle (SILC), is used for all work and is meant to embody Capability Maturity Model® for Software (SW-CMM®) Level 3 processes.

In 2001, the organization decided to run a pilot Personal Software Process^{SM} (PSP^{SM})/Team Software Process^{SM} (TSP^{SM}) technology insertion program by training developers in PSP and launching TSP teams. The motivation for introducing these processes was to supplement SILC with personal/team processes that encouraged process improvement for the individual developer.

The SILC process is organized in phases consisting of Planning, Analysis, Design, Implementation, and Deployment. The organization's business model requires a proposal for each project, which must be approved by both the customer and management. After approval, the SILC process is enacted throughout the life of the project. The TSP projects launched so far have, with the exception of one case, started with the analysis phase. Figure 1 shows the overlay of the TSP process on the SILC process.

Early Experiences Using TSP

Since the initial pilot in 2001, a significant number of development staff has been trained in PSP and TSP. Fifty-six developers have completed the 10-day PSP class, and 44 managers and non-developers have completed the two-day Introduction to PSP class. The TSP process requires a multi-day project launch, which produces an overall project plan and a detailed next phase or cycle plan. A relaunch occurs at the end of each phase or cycle to develop the next cycle or phase detailed plan. The first TSP launch occurred in February 2002; currently, a total of eight projects have used TSP with eight launches and five relaunches.

During the early launches with TSP, there was a lot of hesitation by teams about using TSP in addition to the mandated SILC process. A great deal of discussion centered on whether TSP could replace SILC outright since TSP already has a defined process; the argument went further on the congruity of SILC roles and TSP roles. SILC defines a set of product engineering roles such as analyst, designer, builder, and database administrator. The TSP defines roles that are a combination of product engineering roles and project/process management roles such as design manager, implementation manager, planning manager, process manager, quality manager, support manager, and test manager. Aligning these two role sets consumed a fair amount of time and team energy.

The Software Engineering Institute prototype TSP tool also became a major point of contention: the tool did not provide the user-friendly aspects developers have come to expect. Phases in the TSP tool did not align readily with SILC phases, and users were not able to change them. Even the TSP earned value system came under attack from some project leads who had attended Project Management Institute (PMI) training as different from the PMI and Department of Defense (DoD) definition of earned value. The TSP earned value system focuses on effort and does not address cost, while the PMI and DoD version of earned value track cost and budget burn rate. In addition to the information on effort expended provided by TSP, project managers have to monitor the organizations' financial management system to have an accurate picture of total project costs.

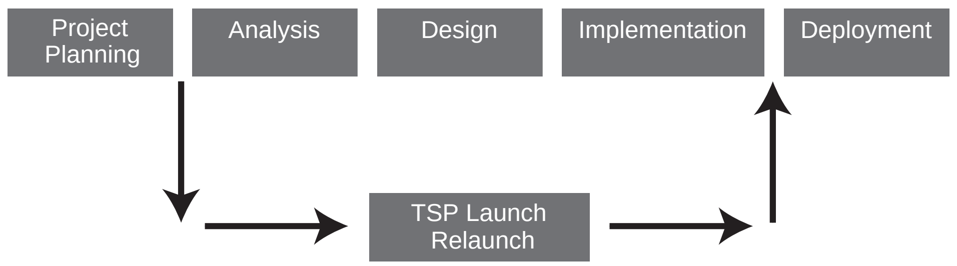
Based on these early experiences, it was not clear whether these struggles would result in the outright rejection of TSP as an organizational process. Two developments led to a more favorable view of TSP use within the organization. Management issued a policy – without specifying use of the TSP – that required all projects to report their status based on facts and data. The policy provided an addendum to the SILC glossary, defining terms such as plan and actual task hours, percent work complete, unplanned task hours, etc. This policy changed the nature of the TSP insertion from a *push* from above to a *pull* from managers and project leads that needed a way to satisfy the policy. Additionally, TSP began to be viewed by project leads as a useful way to structure and run their projects.

Overall, it has taken some time to build support for TSP among management and practitioners. However, each project provided some valuable lessons learned and also provided greater visibility into some of the organization's issues. Lessons learned from the early experiences are described here.

Lessons Learned From the Initial Experiences Using TSP

Most participants in the early TSP project saw the potential of this methodology, if some of its concerns could be addressed. The use of TSP highlighted several process improvement areas for the organization as a whole. While the launch meet-

Figure 1: *Insertion of TSP into SILC Process*



ing scripts and requirements engineering improvements were specific to TSP, project time accounting and the number of projects per developer applied to the organization as a whole.

Launch Meeting Scripts

First, everyone agreed that the TSP launch process requiring extended discussion on project issues between team management and customer, and among team members was highly beneficial. One developer commented, "The most beneficial aspect of this process was the TSP launch.

"The launch gave us the opportunity to step away from work for a few days, think about what the project required, and plan for the project accordingly." The launch scripts were altered to include SILC-mandated management reviews at the end of each launch. Stakeholders, including funding sources, end-user representatives, and project management were able to see the results of the launch and were presented with an accurate plan of what the project would accomplish in the next three to four months.

Requirements Definition

Through the use of both SILC and TSP, it was discovered that the feature-level requirements typically found in a SILC proposal did not provide enough detail for systems analysts and designers to create their TSP plans. Some early TSP launches were suspended while the team worked requirements to the point where the team had a common mental model of what was needed from the system. It was decided that the minimum knowledge of requirements and systems characteristics needed was equivalent to the information required by a concept of operations document (IEEE Standard 1362-1998). A concept of operations document is now required for TSP launches.

Time Tracking

The idea of time logging was initially not well received by the practitioners. It was not clear to them how this data would be used and some worried that management might use the data to judge their performance. Over time, teams have come to realize that the increased visibility provided by this data not only helps management, but also the teams have a better understanding of project progress.

Task Time

Task time is defined by TSP as time spent on project-related tasks. All other activities such as attending meetings, e-mailing, taking a break, etc. are not considered task time. The organization did not have the metrics to know the amount of task time available for developers. When the TSP coach suggested starting with 20 hours per week for planning purposes, one manager's reaction was, "What will we do with the other 50 percent of the time?" After a few projects, data showed that 20 hours per week was not possible due to other duties and the dynamic nature of the environment.

Multiple Projects Per Developer

Developers were divided among several projects; some developers were divided among as many as four or five projects making task time availability for the TSP project only 10 percent (two hours per week) or 20 percent (four hours per week) using the measure of 20 hours of available task time per week. Data showed that such a limited resource assignment method is inefficient. The practice has now been changed to allocate each developer to a maximum of two projects.

Moving Up to the Next Level

As mentioned earlier, last year the Information Systems Development Center mandated that projects report their status with facts and data. Guidelines for a reporting system and a glossary of terms were published to help management and practitioners understand the policy requirements. When a new manager (co-author of this article) took over the management of a department within the Development Center, one of 10 such units, he was faced with the mandate of data reporting.

The new manager made the tactical decision to make TSP available to any team that requested it. So far, three teams have selected the TSP as their method of choice to manage their data projects. The experience this time around is markedly different from our initial experiences with TSP. In what follows, we describe the policy for managing projects with data and our new experiences with TSP. We also describe the behavior of a TSP practitioner, and the TSP reporting system that provides status reports that one can act on. Finally, insights are offered on how we use the data to manage a portfolio of projects.

A Typical Day in a TSP Practitioner's Life

The TSP practitioners in the group noted previously exhibit very different behavior patterns from those in the past. They like the model of personal and team planning as opposed to a plan handed to them by a project lead. They take ownership of the project and not just of the tasks assigned to them.

A typical day for a TSP practitioner is partitioned into two areas: the time spent on tasks related to the TSP project and time, and the time spent on other activities. TSP practitioners keep track of project task time to the minute using a time log. Individual developers use a defined process to develop software and record time based on phases of their personal process; it is rich with personal reviews for early defect removal.

A typical TSP developer's personal process would have phases such as *design*, *design review*, *code*, *code review*, *compile*, and *test*. All these steps are measured in terms of time spent and defects injected or removed. By collecting the three basic measures of effort, defect, and size, and by recording task completion date, a host of metrics is available to the developer. These metrics help the developer manage work, and compare actual work with planned work at a personal level. An example of the type of metrics derived from the three basic core measures is the dashboard style data shown in Table 1.

The data in Table 1 is from a developer's plan in week three of an eight-week project. It shows that, so far, effort estimation has been twice that of actual effort needed for completed tasks. As a result, it shows that the project is ahead of schedule: 51 percent complete compared to the planned 30 percent complete. Clearly, this is good information to have at an early stage, as there inevitably is someone else who may be behind in his plan. During the TSP weekly meeting, one of the activities is load balancing when the data indicates the need.

The practitioner submits his/her personal plan to the planning manager (a member of the team) for consolidation on

Table 1: *A Developer's Weekly Status Report*

Weekly Data	Plan	Actual	Plan/Actual
Project hours for this week	20.0	22.6	0.88
Project hours this cycle to date	33.0	28.0	1.18
Earned value for this week	19.2	36.3	0.53
Earned value this cycle to date	30.0	51.1	0.59
To-date hours for tasks completed	49.7	25.7	1.93

a weekly basis. The planning manager creates a consolidated plan for the team leader to use for weekly status reporting, as well as for periodic management reporting. This consolidated plan allows the team to know the true status of a project and to take corrective actions when necessary. A similar view is available for the team to analyze the status of the project.

TSP Project Reports

The primary outcome using the TSP model is project reports. Periodic review of project deliverables and regular status reports are built into the TSP process. These reports need to be accurate, actionable, and based on verifiable project data. In essence, these status reports can be thought of as *dials* on a development manager's equivalent of a pilot's instrument panel that show the status of many projects. A continual scan of the dials should reveal unfavorable project trends and allow for their early correction.

Status reporting for a software development project – the dials – provides a quantitative answer for some typical status questions such as the following:

1. How much of the project is complete at this time?
2. Given the progress at a given point in time, when will the project finish?
3. How much time has the development team been able to devote to project tasks?
4. Has the team spent substantial time on unplanned work?

Two charts can be created that will quickly provide answers to these questions. The first chart uses a measure called *earned value*. Earned value is one way to determine how much of a project is complete, and predict when it might finish. This measure is calculated by comparing the planned hours for completed tasks to the total planned hours for a project. A couple of comments on the earned value calculation: It is important to stick to the planned hours when calculating earned value even though more *actual* hours were needed to complete the tasks involved. Also, the hours for completed tasks should only include those tasks that are 100 percent complete. In other words, a task is done or it is not done. This prevents counting tasks that are perpetually 90 percent complete and thereby overstating a project's progress.

Figure 2 shows an earned value chart for a project. Notice that the *actual* earned value, or the sum of the hours for completed tasks, is shown against the *planned* earned value. In a straightforward way, these two lines show how well the project is tracking against the original plan. Ideally, these two lines will overlay each other, thus signifying a project that is on plan. As is often the case, however, actual earned value will lag behind planned earned value. When this lag is 10 percent or less, the cause may lie with major tasks that will soon finish or project conditions that will respond to minor corrections.

When actual earned value deviates by a larger amount, stronger corrections may be needed. One of the causes of this deviation often is unplanned work. Work that was not anticipated and included in the project's planning, but was deemed necessary. A line plotting the hours for unplanned work is included in Figure 3 to track this phenomenon.

From looking at the earned value chart in Figure 2, overall project progress is easily determined. Since only completed tasks are counted, the actual earned value readily shows that the project is 40 percent complete. An expected time of completion can also be surmised from this data. Given a constant earned value rate for the project, a 10 percent variance on a 52-week plan would cause the project to finish approximately five weeks after the planned finish date. Alternatively, the development manager could attempt to recover and catch the project up by requiring overtime or assigning additional staff.

The planned and actual team hours shown in Figure 3 can often provide clues on why the project's earned value is lagging. In this case, the team's actual hours are behind what was planned; for some reason, team members were not able to devote as much time to the project as expected. This could result from an estimation error – typical software engineers can expect to put in 20 hours of a 40-hour

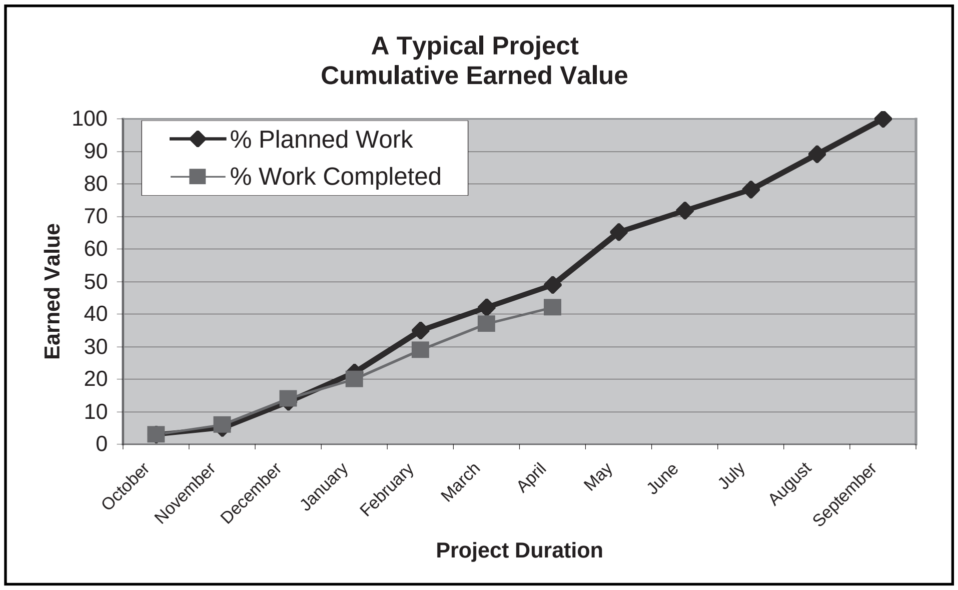
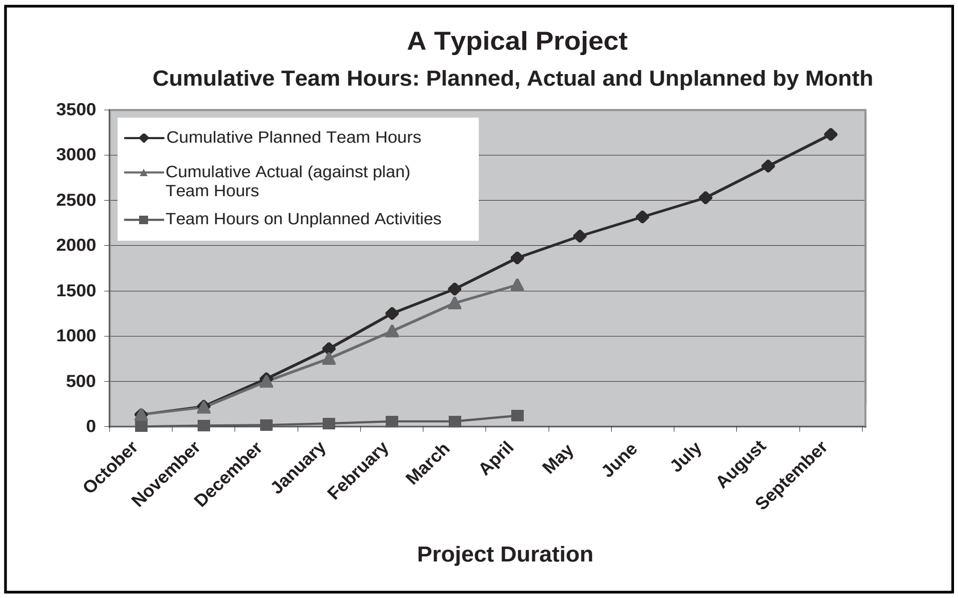


Figure 2: *Planned Versus Actual Earned Value*

Figure 3: *Planned Versus Actual Hours*



week on project work. The rest go to meetings, ad-hoc assignments, etc. This 50 percent productivity factor can be lower depending on the environment. Other projects with competing priorities may have pulled team members away. Also, team hours used on unplanned activities are worth tracking and monitoring; this is the time spent on project-related work that was not anticipated or included as part of the project plan. Often these hours can be substantial, and one can see an increase in unplanned activities that are usually accompanied by a corresponding decline in actual team earned value.

Stepping back and taking a look at both charts, the development manager can draw the following conclusions. This project started to deviate from estimates as early as January. Progress appears to be hampered by the amount of time the team is able to apply to the project; if no action is taken, the project will miss its completion date. This is an early warning for the development manager to talk to the team lead and project members, ask some pointed questions, and develop a much richer picture of the project's status and obstacles.

In addition to schedule data, the TSP reports provide a rich set of quality data and an early warning system related to the quality being built. Although the previous two charts help us to track schedule progress, we also need to know that the product being built is of acceptable quality so it will not get bogged down during the test phase – a typical scenario in many software projects.

One *dial* that TSP does not provide the development manager with is projected and actual projects costs. There are a number of reasons why TSP-supplied data is not sufficient for providing financial status. TSP plans and measures activity in terms of available hours, and even though the typical software engineer has 20 hours available per week, he or she must be paid for the full week. There is also no provision for tracking common project expenses such as training and purchases. A scheme for loading available hours to accommodate these costs seems possible but has not been attempted in the projects in which either author has been involved. That said, TSP's performance and schedule have proven very useful at Sandia, and the use of the organization's financial system to help provide the overall project status has not been burdensome.

Managing Project Portfolio

The *secondary outcome using the TSP model is the ability to manage a portfolio of projects.* As mentioned before, Sandia's Information Systems Development Center has many projects at any given time; currently only three projects are using the TSP. However, development managers must report status on all projects in their portfolio to their superiors and funding sponsors. Understandably, these stakeholders want visible and objective measures of progress for the projects involved. Again, the desire is to stay abreast of the development portfolio and intervene should the development manager need assistance keeping projects on track. Whether that report is a scorecard or takes another format, accurate data that reflects the true state of affairs is needed.

The difference between two types of projects (TSP vs. non-TSP) is like day and night when one tries to prepare scorecard reports. The TSP projects have all the data collected as part of their process; with non-TSP projects, the project lead must work much harder to collect time worked, the numbers are often less accurate, and it is harder to determine the true status of the project. After having experienced both reporting systems, undoubtedly the TSP satisfies management by data requirements without any additional effort. As mentioned earlier, the developers readily see the value of data collection from their personal work processes, and when meeting weekly as a team.

Summary and Conclusion

We do not intend to leave the impression that all issues are resolved, and that TSP is being used seamlessly in this organization. The size issue raised during the first launch has not gone away. The organization uses function points to measure project size by personnel outside of the project. The TSP teams are not completely trained to use function points for their size estimates and are resistant to using lines of code as their measure. Developers reason that since they are working in a multi-tier architecture environment using multiple languages, using lines of code does not make sense for their application environment. The collection of defects at the personal level is just getting started, and defect-counting standards for design have been created. Developers have been hesitant to record defects, fearing that this information may fall into the wrong hands and be used for evaluation purposes. Not all teams have chosen to use TSP to respond to the management-by-data mandate.

On the other hand, software projects using the TSP have experienced a number of successes. The three TSP projects exhibit good project control and tend to need only minor corrections because problems are detected early. TSP meeting scripts and the guidance of the TSP launch coach have been an excellent way to support new development project leaders. The format of TSP team meetings and use of its roles has increased the teams' sense of ownership of the work and process. We look forward to improved quality and performance as metrics are collected from all members of the development team and used to improve team and personal processes. As mentioned earlier, the rigorous timekeeping of these TSP projects provides excellent project visibility. Since this data comes directly from those doing the work, status reports derived from this information are fairly objective and can provide good insight into the progress of a project.◆

About the Authors

Ray Trechter, Certified Software Development Professional, is a software development manager at Sandia National Laboratories. In addition to managing software development projects, Trechter has worked in the areas of software architecture, software process improvement, and as a developer of distributed systems.

Sandia National Laboratories
MS 0661
Albuquerque, NM 87185
E-mail: ratrech@sandia.gov

Iraj Hirmanpour is principal of AMS, a software process improvement firm and a Software Engineering Institute Personal Software Process^{SM} (PSP^{SM}) and Team Software Process^{SM} (TSP^{SM}) transition partner. Hirmanpour is a PSP/TSP consultant and trainer on the transition of technology to organizations.

AMS, Inc.
421 Seventh ST NE
Atlanta, GA 30308
Phone: (404) 394-2028
E-mail: ihirman@earthlink.net

Personal Earned Value: Why Projects Using the Team Software Process Consistently Meet Schedule Commitments

David Tuma
Software Process Dashboard Initiative

David R. Webb
Hill Air Force Base

*Data from dozens of projects using the Team Software Process*SM *(TSP*SM*) provide powerful proof of success at consistently meeting cost and schedule commitments. While disciplined engineering and high quality processes are important factors contributing to these successes, mathematical analyses of project data indicate that the most important factor is the proper management of earned value techniques at the team member level. In fact, this practice – unique to TSP teams – can produce a 10-times reduction in schedule variance by properly balancing team workload using personal data.*

Projects using the Team Software ProcessSM (TSPSM) developed by the Software Engineering Institute (SEISM) have a phenomenal performance record, especially in terms of meeting schedule estimates. As noted by Watts Humphrey in this issue of CROSSTALK, current industry data show that more than one-third of all non-TSP software projects still fail [1]. In stark contrast, data gathered by the SEI from 20 TSP projects in 13 different organizations show that these TSP teams missed their schedules by an average of only 6 percent and had a very narrow schedule variance range, from 20 percent earlier than planned to 27 percent later than planned [2].

Why do projects using the TSP succeed at meeting schedule commitments so often and so well? Conventional wisdom suggests this world-class performance is due to two reasons: (1) TSP software engineers have become experts at using historical data to produce highly accurate estimates; and (2) TSP projects employ quality methods that drastically reduce or even eliminate defects found in later process phases (such as integration, system, and acceptance testing), rendering these typically volatile development activities consistent and predictable.

Years of real-world TSP project experience suggest that TSP's approach to earned value planning and tracking is also a significant factor in meeting schedule estimates. In fact, while the factors listed above are of great importance, our analysis indicates that the management of earned value at the team member level is more important than both these factors combined. To understand why, it is helpful to compare traditional earned value project management to the approach used in the TSP.

Traditional Earned Value Planning and Tracking

Many projects use a method called *earned value* to plan and track progress. At the beginning of a project, teams using earned value will define a list of high-level project tasks, and estimate the time each task will require. As shown in Figure 1, a predicted completion date for each task can be estimated by determining when the project will have expended the requisite effort or Budgeted Cost of Work Scheduled (BCWS).

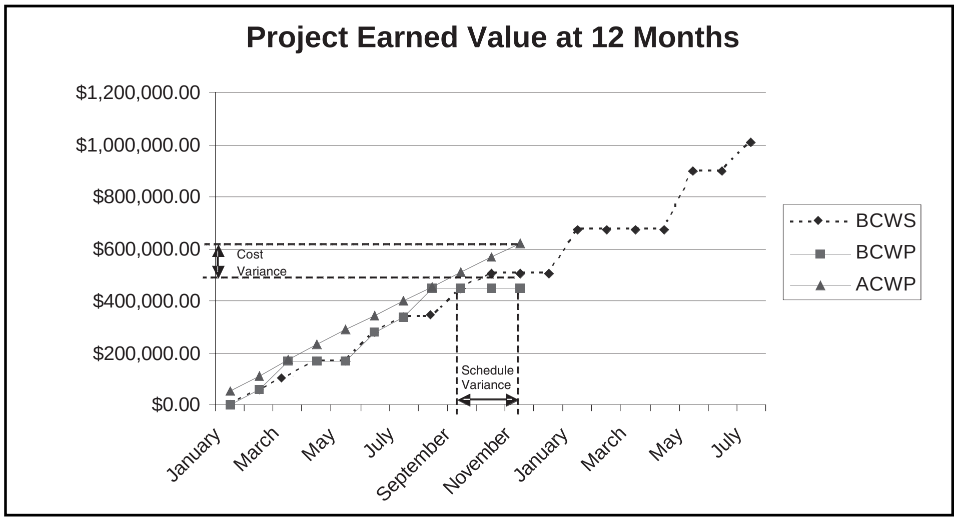
The earned value method then assigns each project task a *value* based upon its estimated cost or effort. As each task is actually completed by team members, the project *earns* the originally estimated value for the task; this is called the Budgeted Cost of Work Performed (BCWP). The real cost or effort to complete the task is also tracked as the Actual Cost of Work Performed (ACWP). The combination of these three values, arranged according to planned (BCWS) and actual (BCWP and ACWP) schedule performance, allows projects to determine both cost and schedule variances from their plan.

This traditional earned value technique, while an effective tool, is often incomplete because it is planned for the completion of high-level project tasks (such as overall design, code, and test), is tracked only at the project level, and is reviewed only monthly. Since these large tasks often require more than one month to complete, there is a high likelihood of one or more *zero work* or *flat line* zones on a traditional earned value plan. (Note the BCWS for October to December and January to April in Figure 1.) Within these zero work zones, earned value metrics provide no insight into project progress; this can mask serious problems for months at a time. In Figure 1, a serious scheduling problem that was first encountered by the project in January does not show up on the earned value chart until May.

TSP Earned Value Planning and Tracking

TSP teams create and use earned value plans very differently from traditional teams. At the beginning of a TSP project, the team conducts a *launch* meeting. During the initial launch, tasks are defined at a very high level and estimated using gross measurements such as lines of code per hour. A rough plan is drawn up using these high-level estimates to determine an

Figure 1: *Traditional Earned Value Plan*



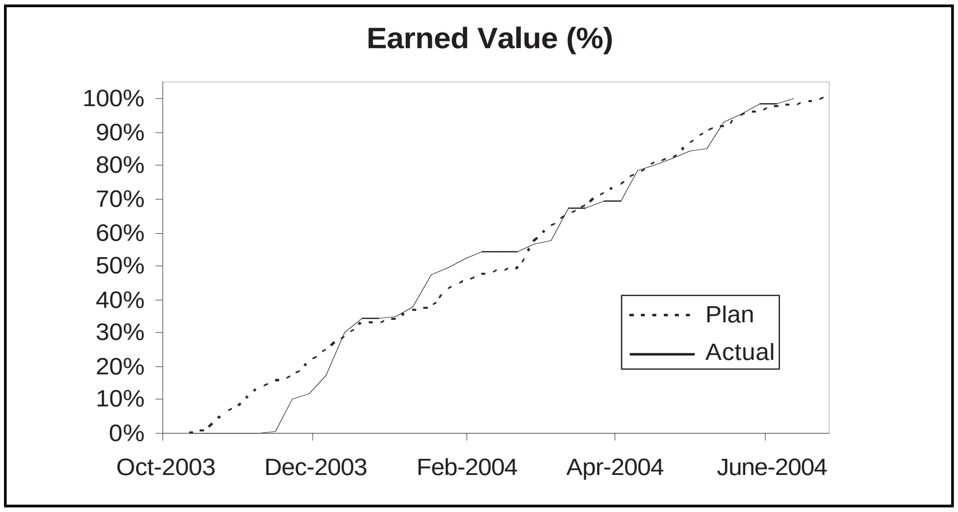


Figure 2: *TSP Earned Value Plan*

idea of the schedule the project will require to complete the assigned workload. Once this is complete, the TSP team subdivides the work into three-to-four month phases, breaking the first phase into detailed subtasks of fewer than 10 hours each. These tasks are assigned to individuals on the team, and personal earned value plans are created for each team member. These personal plans are then consolidated to create a team plan.

Once the launch is finished, the individuals immediately begin working to complete their assigned tasks, tracking their progress using their personal earned value plans. One of the SEI's stated entry criteria to launching a TSP project is that team members are trained in the Personal Software Process^{SM} (PSP^{SM}). Among other things, PSP students learn how to estimate in pieces, break down their personal work into measurable tasks, and gather minute-by-minute data on their progress to create detailed earned value plans. As a result, TSP teams have continuous access to real, measurable data on task completion, duration, and cost (effort). So, rather than the stair-step, month-to-month plan shown in Figure 1, TSP teams produce and live by much more detailed earned value plans like the one shown in Figure 2. The *smoother* look to this plan is due to a much higher granularity of measurement than is practiced or even possible on traditional earned value projects. Note that there are basically no zero work zones in Figure 2, even during the weeks of Christmas and Independence Day!

Using this level of detail, the team holds weekly meetings where they review progress against the personal and team earned value plans. These weekly reviews include an examination of the forecast completion date; if the forecast differs significantly from the plan, the team produces corrective action plans to address the variance. Since individual team member data is available to supplement the rolled-up team measures, it is immediately obvious to TSP teams which tasks and team members are ahead of schedule and which need assistance. This information gives the team members the insight, on a weekly basis, to adjust task assignments, renegotiate functionality with the customer, or replan work to keep the project on track.

The Significance of Personal Earned Value

This individual earned value tracking methodology provides a very sound basis for planning and managing a team software project. Three fundamental behaviors are the key to this management approach:

1. Fine-Grained Estimation (subdividing project tasks before estimating).
2. Forecast Tracking (monitoring forecast cost and completion date).
3. Workload Balancing (reassigning workload between team members).

Although these seem to be fairly common-sense behaviors, they are radically affected by earned value metrics tracking at the personal level. At first glance, the first bullet would appear to be the most important of the three practices. Most TSP practitioners would be surprised to discover that the bullets are actually listed in *increasing order of importance*. Workload balancing, in fact, has the *most* significant impact by far on the reduction of project schedule variances. To understand why, it is helpful to examine these behaviors in light of a few simple and well-understood statistical phenomena.

Estimating Basics[1]

Estimates, of course, are never perfect, and estimating errors are inevitable. The quality of a series of estimates can be characterized by two metrics: precision and accuracy. Estimating precision is the concept most people think of first. For example, an estimate that falls within 5 percent of the final value could be described as very precise. Most organizations have a strong business need to minimize cost and schedule overruns and overestimates; consequently, they focus on reducing the size of their estimating error.

Estimating accuracy, on the other hand, describes the bias in a series of estimates. If an organization were to consistently underestimate project cost, their estimates would not be considered very accurate. An even balance between overestimates and underestimates would characterize an accurate estimating process.

When estimating a large project, it is common to begin by breaking the work down into smaller tasks, estimating those tasks independently, and summing the results. Outlining tasks in greater detail can generally produce a more precise final estimate. Although this practice intuitively seems to be correct, statistical concepts explain this mechanism mathematically. Imagine, for example, you have a sequence of independent estimates for individual subtasks, like those in Table 1.

In Table 1, subtask 1 is estimated (with 70 percent certainty) to require between 75 and 125 hours, with 100 hours being the most likely cost. The individual task estimates would be summed to produce a total estimate of 550 hours. The ranges, however, cannot simply be summed (which would produce a range of ±125 hours). If these estimates are accurate (balanced between underestimates

Table 1: *Example Subtask Estimates and Ranges*

Task	Estimate (Hrs)	Range (70%)
Subtask 1	100	±25
Subtask 2	160	±30
Subtask 3	90	±20
Subtask 4	200	±50
Total	550	±67 (not summed)

and overestimates), it would be very unlikely for the actual project to complete *every* subtask at either the low or the high end. As a result, it can be assumed that over- and under-estimates will partially cancel each other out. Statistically, the estimated range for the overall project can be calculated by squaring each range, summing those values, then taking the square root. This approach yields a prediction range of ±67 hours and makes the range around the sum of the estimates *considerably* tighter than the range around the estimate of each individual task.

This important concept is the basis for many industrial-strength estimating practices (including the cost estimation practices in the TSP). Although its application to cost estimation is well known, its implications for schedule estimation are much more profound (as will be explained).

Fine-Grained Estimating Defines 10-Hour Subtasks

When planning work for the next three- to four-month project iteration, TSP project teams create a plan that divides project work into subtasks of approximately 10 hours each. This behavior can be quickly understood as an example of the *estimating precision technique* described in the previous section. In practice, however, TSP teams rarely generate *independent* estimates for each subtask, which was a basic assumption for the sum-squares range calculation. Instead, larger tasks are estimated, and historical percentages are used to automatically subdivide those tasks into smaller parts. As a result, the individual estimates are not independent, and do not fully benefit from the statistical mechanisms described.

As mentioned, TSP teams require software engineers to be trained in the PSP. While it is true the PSP teaches engineers well-defined, statistically based methods for producing accurate estimates, TSP teams rarely use those methods to produce team plans. The PSP PROxy Based Estimating (PROBE) method requires abundant historical data at the personal level; teams rarely have access to that kind of data when they first launch. Even after archiving considerable data, teams generally use historical averages based on team-level metrics to produce their plans. Although the co-authors have collectively participated in more than a dozen TSP launches (including projects listed in the SEI studies cited earlier), we have actually never been part of a TSP launch that used PROBE methods for team planning purposes.

Task	Planned Hours	Actual Hours	Estimating Error %
Subtask 1	4	0.8	80%
Subtask 2	3	1.7	43%
Subtask 3	1.2	4.6	-283%
Subtask 4	10	4.3	57%
Subtask 5	5	7	-40%
Subtask 6	5	14.5	-190%

Note: Estimating Error % is calculated as (Plan-Actual)/Plan

Table 2: *Example Task Data From Hill Air Force Base TSP Project*

Furthermore, the need to produce such detailed estimates early in the project, with limited available estimating time, typically results in estimating errors that are much larger than those measured by engineers during the PSP training course. Consider the excerpt of data in Table 2 from a recent TSP project at Hill Air Force Base.

These subtasks, chosen at random, demonstrate the *significant* estimating errors that occur when work must be broken down to the 10-hour level during an initial project launch. The histogram in Figure 3 shows the estimating errors for all subtasks completed by the project during a 12-month period.

As Figure 3 indicates, estimating errors at the subtask level are large and widespread. More than two-thirds of the subtasks in this project were misestimated by 50 percent or more. This metrics trend is not unique to this project. As a result, the fine-grained estimating performed during a TSP launch rarely enables teams to see the cost estimating precision benefits that would be projected by a sum-squares range calculation, or by the estimating accuracy improvements described in PSP studies.

Without question, many TSP teams *are* able to finish projects with very small cost variances. These achievements, however, are not generally accomplished with the statistically precise estimating methods taught in the PSP course. Instead, TSP teams are able to manage cost variances with mid-course corrections, enabled by the forecast tracking behaviors described in the next section.

In fact, the project whose data is illustrated in Table 2 and Figure 3 was *highly* successful, completing with a cost variance of 17 percent (under planned cost) and schedule variance of only 2 percent (ahead of schedule). These phenomenal results were explained by a team member, who said, "Our project succeeded [on cost and within schedule] because *we made* it succeed." The subtask data indicate that these results were not due to precise, fine-grained task estimates. Instead, it was accomplished by diligent forecast tracking and workload balancing.

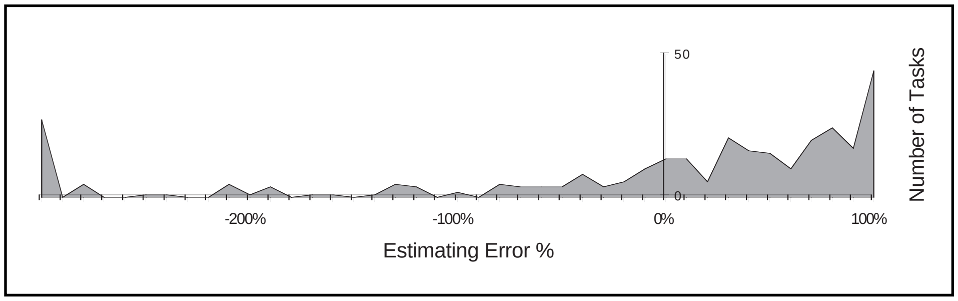
Fine-grained estimating, then, does not carry the full statistical significance suggested by the conventional wisdom. It does, however, provide an important tangible benefit: Defining tasks at the 10-hour level allows earned value progress to be tracked weekly. This helps the team to maintain a focus on continual progress, and facilitates early detection of problems.

Forecast Tracking Reveals Biases Early

Early detection of problems is the primary goal of forecast tracking. As described earlier, TSP teams collect earned value metrics daily and review them weekly, and produce corrective action plans when forecast cost and forecast completion dates differ significantly from the baseline. If these corrective action plans are unsuccessful, the team will quickly escalate the issue to management and to the customer.

Collecting earned value metrics at the personal level significantly increases the granularity of the resulting metrics, enabling TSP teams to discover cost and

Figure 3: *Histogram of Task Estimating Errors From Hill Air Force Base TSP Project*



Scenario	Team Member A	Team Member B	Unbalanced	Optimized
1	Early	Early	Early	Early
2	Early	Late	Late	On time
3	Late	Early	Late	On time
4	Late	Late	Late	Late

Table 3: *Four Workload Balancing Scenarios*

schedule discrepancies much earlier than usual. Early knowledge of these discrepancies allows the team to renegotiate scope and/or alter their technical direction, which can facilitate a significant reduction in final cost variances.

Although teams will strive for accuracy in their estimating process, significant estimating biases are still quite common. Forecast tracking provides a way for teams to discover these biases early when there is still time for the project to recover.

Workload Balancing

Workload balancing is the act of reassigning tasks from overburdened team members to under-tasked team members. The goal of workload balancing is to produce a plan in which all team members finish their assigned work on approximately the same date. Workload balancing is uniquely enabled by earned value tracking at the personal level.

Of the earned value management behaviors described, workload balancing is by far the most important. To understand why, it is helpful to consider the difference between two metrics:

- **Unoptimized Forecast Completion Date.** The date the project is forecast to complete, if progress continues at historical rates, and if team members perform tasks as assigned in the current project plan.
- **Optimized Forecast Completion Date.** The date the project is forecast to complete, if progress continues at historical rates, and if tasks are reassigned to balance the workload optimally.

With earned value schedules for each individual on a team, it is simple to calculate these two metrics for the overall team. The optimized date can be calculated simply by summing up data values to the team level, and using traditional earned value equations[2]. The unoptimized date can be calculated by looking at the personal schedules and seeing who finishes last.

Examining a very simple case can illustrate how these forecasts differ. Consider a team with only two individuals, and consider the various scenarios (shown in Table 3) where the individuals finish 20 percent early or 20 percent late.

Scenarios 1 and 4 show the presence of a consistent estimating bias. Workload balancing does not help in these scenarios, but fortunately these problems can be detected and corrected via the forecast tracking activity described in the previous section. Scenarios 2 and 3 show the simple effect of workload balancing; in these scenarios, the projects would finish late, but workload balancing helps them finish on time instead.

Table 3 makes clear a very simple observation: Workload balancing allows schedule variances to additively cancel each other out. This is an incredibly important point because it allows project schedule variances[2] to benefit from the sum-squares reduction described earlier.

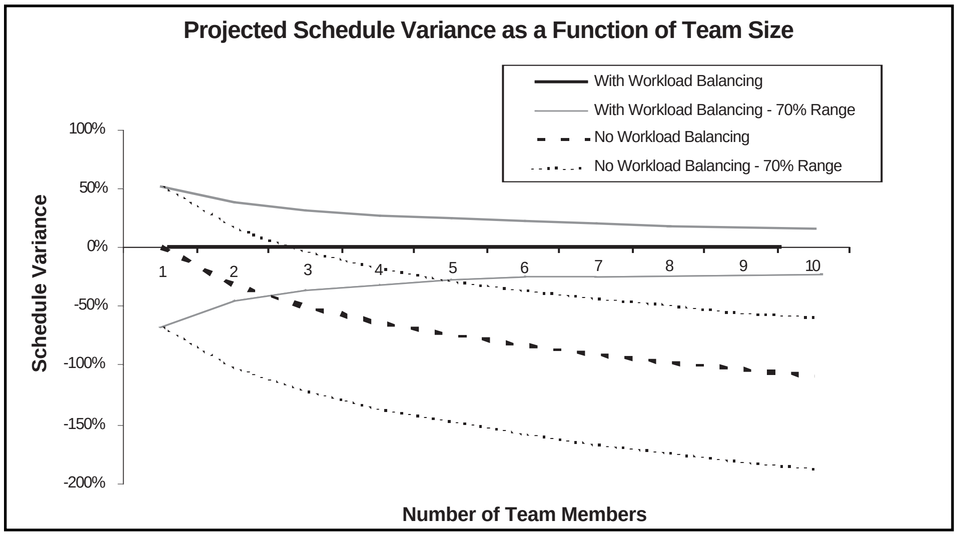
It is also possible to make this observation mathematically. Consider a project team with two individuals. Both individuals estimate, with 70 percent certainty, that they will complete the work assigned to them by the end of September. If the workload is not balanced, what is the likelihood that the overall project will finish by the end of September?

Since the workload is not reassigned, we can observe that the project will complete when the last person finishes. Since there is a 70 percent probability that Team Member A will finish by the end of September and a 70 percent probability that Team Member B will finish by the end of September, a simple calculation (0.70 x 0.70 = 0.49) indicates that the overall probability is only 49 percent. This probability drops exponentially as individuals are added to the team: with eight team members, the projected likelihood of project completion by September 30 drops to less than 6 percent. This dismal percentage is due to the fact that a problem encountered by any individual can affect the project's completion date.

Of course, a confidence level of 6 percent is not useful when reporting forecast completion dates to management or to the customer; 70 percent prediction ranges are more in line with expectations. To estimate the project completion date with a team of eight people with 70 percent certainty, you need the 95 percent ranges for each individual! This graphically illustrates why most projects don't finish on time.

In fairness, the optimized and unoptimized metrics here are extremes. Even with the best workload balancing, a TSP team will never be able to perfectly optimize their plan; nevertheless, they are often able to come *very* close. And even on a non-TSP team, some workload balancing is likely to occur. But tracking earned value at the personal level has an undeniably significant impact on the effectiveness of workload balancing. By applying forecast tracking to the earned value plans of each individual, teams are able to notice imbalances early and reassign tasks that have not been started yet. In contrast, most non-TSP teams do not discover imbalances until late in the project. This awareness often comes too late to meet the originally committed completion dates, and the need to transfer knowledge from the overcommitted individual to other team members catastrophically impairs productivity [3]. For the

Figure 4: *Simulation-Projected Schedule Variance as a Function of Team Size*



Note: The schedule variances shown are based on a cost estimating error of ± 50% and a weekly task time estimating error of ± 25% (both for 70% certainty ranges).

entire period of the project before the imbalance is noticed, individual team members will have been pacing themselves, consciously or unconsciously. An under-tasked team member, seeing that they are comfortably meeting the dates required of them, will have most likely devoted a significant amount of time to non-project-essential tasks, unaware that their co-worker needed help. That time spent on non-essential work can never be recovered. In contrast, continual workload balancing helps to establish a shared level of urgency among team members.

Analysis With Numerical Methods

These simple mathematical analyses illustrate how unoptimized forecast dates become exponentially less reliable as individuals are added to a project. But when *a* workload is balanced based on personal earned value metrics, schedule overruns and underruns are able to cancel each other out, resulting in significantly smaller schedule variances for the overall project.

Using numerical methods, the authors of this article have succeeded in demonstrating this fact mathematically [4]. The results were striking: All other factors being equal, workload balancing predicted schedule variances that were orders of magnitude smaller than the schedule variances for unbalanced work. Figure 4 illustrates the results: With no workload balancing, a project is more and more likely to finish behind schedule as team size grows. In contrast, a project that balances workload optimally has more opportunities for workload balancing as team size grows, increasing the likelihood that the project will finish on time.

This analysis seems to suggest that workload balancing, enabled by personal earned value tracking as practiced in the TSP, can by itself account for a *90 percent* reduction in the schedule variance of a project. These incredible results suggest that personal earned value tracking is *predominately* responsible for the tiny schedule variances seen by TSP projects.

Conclusions

The TSP includes many high-maturity behaviors that help teams produce superior results. While nearly all of these behaviors affect a team's on-time schedule performance, numerical analysis seems to indicate that proper application of earned value at the personal level is the largest single factor enabling the tiny schedule variances seen by TSP teams.

Curiously, engineers only receive about an hour of earned value training in the PSP class, and they do not typically use these techniques during the course. Most engineers do not actually experience the practical application of earned value until they start their first TSP project (and historically there has not been any extra earned value training at that point)[3].

These facts seem to beg these questions: "Could personal earned value tracking be used alone, without other PSP/TSP techniques, by teams in otherwise mature organizations?" " If so, what are the critical enabling success factors?" "What results might be expected?"

A six-sigma design of experiments seems warranted. While one would not expect to see the quality successes produced by TSP projects, the authors of this article feel that targeted earned value training and proper management support may allow non-TSP teams to enjoy significant cost and schedule benefits.◆

References

1. Humphrey, Watts S. "Why Big Software Projects Fail: The 12 Key Questions." CROSSTALK Mar. 2005 <www.stsc.af.mil/crosstalk/2005/03/0503 Humphrey.html>.
2. Davis, Noopur, and Julia Mullaney. The Team Software Process^{SM} (TSP^{SM}) in Practice: A Summary of Recent Results. Pittsburgh, PA: Software Engineering Institute, Sept. 2003.
3. Brooks, Frederick P. The Mythical Man-Month. 1st ed. Reading, MA: Addison-Wesley, 1995.
4. Tuma, David A. "A Statistical Model for Personal Earned Value, and Implications for Project Planning." 31 Dec. 2004. <http://processdash.sourceforge.net/ev.html>.

Notes

1. This article does not attempt to fully explain statistical estimating methods. It only describes these methods at a high level as background for the following discussion.
2. Variance is used in the project management sense, not the statistical sense.
3. TSP teams could potentially benefit from additional earned value training, to take full advantage of the powerful tools they have at their disposal.

About the Authors

David R. Webb is a senior technical program manager for the Software Division of Hill Air Force Base in Utah, a Capability Maturity Model® for Software Level 5 software organization. He is a project management and process improvement specialist with 17 years of technical, program management, and process improvement experience with Air Force software. Webb is a Software Engineering Institute-authorized instructor of the Personal Software Process, a Team Software Process launch coach, and he has worked as an Air Force section chief, Software Engineering Process Group member, systems software engineer and test engineer. He is a frequent contributor to CROSSTALK and has a bachelor's degree in electrical and computer engineering from Brigham Young University.

OO-ALC/MASM
7278 Fourth ST
Hill AFB, UT 84056
Phone: (801) 777-9737
Fax: (801) 775-3023
E-mail: david.webb@hill.af.mil

David Tuma is the lead developer for the Software Process Dashboard Initiative, creating open source tools to support high-maturity software development processes. He first encountered open source software as a student at the Massachusetts Institute of Technology, and again later as a captain in the United States Air Force. As a strong supporter of open source, Tuma has been developing open source software on his own time for the past 10 years.

Software Process Dashboard Initiative
1645 E HWY 193 STE 102
Layton, UT 84040-8525
Fax: (801) 728-0595
E-mail: tuma@users.sourceforge.net

A TSP Software Maintenance Life Cycle

Chris A. Rickets
Naval Air Systems Command

Team Software Process^{SM} (TSP^{SM}) and Personal Software Process^{SM} (PSP^{SM}) have always been associated with software development, but what about TSP/PSP for software maintenance? This article discusses how TSP/PSP was adapted for use on a software maintenance project, resulting in a new proxy for estimating maintenance activity and the creation of a TSP software maintenance life cycle.

Anyone who has been exposed to Team Software Process^{SM} (TSP^{SM})/ Personal Software Process^{SM} (PSP^{SM}) knows that its life-cycle model is based on software development, but what about TSP/PSP for software maintenance? This article tells how an inexperienced team created a TSP/PSP life-cycle model for corrective maintenance that allowed it to finish well ahead of schedule yielding a 76 percent increase in problems fixed and reducing defects by 38 percent.

Background

The Naval Air Systems Command's (NAVAIR) AV-8B Joint System Support Activity (JSSA) has successfully applied TSP/PSP to new software development and to software maintenance projects for three years.

In February 2001, AV-8B's Joint Mission Planning System (AVJMPS) team began applying the TSP/PSP traditional development life-cycle model to the AVJMPS software development project.

In the spring of 2002, AV-8B's Mission Support Computer (MSC) software engineering team began the H2.0 Block Upgrade maintenance software effort applying TSP/PSP. Their effort under H2.0 Block Upgrade was primarily maintenance. They found that by adapting the TSP life-cycle model to better fit maintenance activity, they were able to isolate and measure rework activities and drive down defect rates while increasing productivity.

Proxies were used to size each problem fix in terms of estimated hours instead of source lines of code (SLOC). There is typically no correlation between the solution of the problem and SLOC. There is a correlation between completion of the problem and time to fix.

Figure 1: *TSP Traditional Development Life Cycle Phases*

HLD	High-Level Design
HLDINSP	High-Level Design Inspection
DLD	Detailed-Level Design
DLDR	Detailed-Level Design Review
DLDINSP	Detailed-Level Design Inspection
CODE	Code
CR	Code Review
CODEINSP	Code Inspection
COMPILE	Compile
UT	Unit Test
IT	Integration Test
ST	System Test

The MSC Software Maintenance Team

The MSC Operational Flight Program (OFP), a real-time embedded program running on a PowerPC processor, provides mission computer functionality for the AV-8B Harrier II+. The OFP comprises 700,000 SLOC, mostly in C++.

The five-member H2.0 MSC software engineering team was relatively inexperienced. Two team members had experience working with the MSC OFP software and the toolset. One of these two was newly promoted to the software lead position having no prior experience in this position. Out of the three new team members, only one was familiar with the toolset, and none of them were experienced in the problem domain. The teams primary tasking was corrective maintenance (i.e., fixing software defects).

It was obvious to JSSA management and the H2.0 software team that the team required startup time to learn the toolset, the MSC OFP software, and TSP/PSP.

Maintenance and Development Life Cycles Are Not a Perfect Match

The MSC software engineering team began development with the traditional development life-cycle phases, shown in Figure 1, which are supported in TSP/PSP training and by the TSP/PSP tool. The team, however, soon realized that this life cycle did not address problems associated with software maintenance.

The team found that the TSP traditional life cycle, as shown in Figure 1, did not include a phase for problem identification. Identifying the problem, recreating the problem in controlled conditions, and identifying the solution are critical activities in the first steps of software maintenance. The ISO [International Organization for Standardization]/IEC [International Electronical Commission] Standard 12207 [1] describes these activities in the Problem Modification and Analysis step of the maintenance life cycle.

Next, the team noted that most software changes did not affect high-level design, and many did not affect detailed-level design. Most changes fell into the corrective maintenance category, resulting from missing or misinterpreted requirements, coding logic errors, or missing source code.

The team also experienced cases where finding the correct solution required iteration through the TSP life-cycle phases. For example, the proposed solution might have an unanticipated side effect, which required iteration back through design, code, and test. There are also cases where determining the complete scope of the problem required multiple probes into the design and code.

The traditional TSP life cycle does not accommodate iteration. While it is possible to add iterations through the phases to one's plan, the original work plan for development activity is based on a single iteration from high-level design through integration test. Consequently all the earned value is associated with the originally planned iteration, and additional iterations show up as unplanned work with no earned value.

The Lite Life-Cycle Model Is Born

TSP teaches that your process must be your own and that TSP can be adapted to fit the way your organization does business. The H2.0 MSC software engineering team adapted the TSP life cycle to fit their team and the maintenance activity.

The H2.0 MSC software engineering team coined the phrases *classic* and *lite* to describe their software lifecycle models. The classic life-cycle model is the traditional TSP development model, while lite life-cycle model refers to the maintenance life-cycle model created by the team. The phases of the lite model, shown in Figure 2, specifically address the shortcomings noted in the previous section.

The lite life cycle adds phases and activities specific to software maintenance, and consolidates phases from the traditional life-cycle model to allow for iteration. The lite life cycle preserves review and inspection activities, although they are less visible and may be reordered from the traditional approach. The following paragraphs explain each of the lite life-cycle phases.

- **IDENT.** During the identification phase, the software engineer works with the systems engineer (SE)[1] to verify the requirements and demonstrate the problem. Early involvement of the SE ensures that the problem as specified in the Problem Report (PR) is correct. Often, the PR describes symptoms rather than identifying the root cause. The desired solution in the PR may also be incomplete. Given an understanding of the problem, the software engineer then identifies the cause of failure condition within the source code and demonstrates the source code problem and failure to a peer. If the peer agrees with the software engineer that the problem has been correctly and completely identified, the software engineer can then move to the INWRK [in-work] phase.
- **INWRK.** The in-work phase encapsulates design, code, and unit test to allow for iteration in the maintenance environment.
- **Design and Design Review.** If a design change is needed, the software engineer implements the changes and reviews the changes using his or her design review checklist. Inspection is deferred to the INSP [inspection] phase, except in cases where a substantial change to the design is required, or where there are special circumstances, e.g., a change to a class in the Common OFP[2]. The decision to delay inspection of small, simple design changes to the INSP phase is based on several factors. First, the cost of putting together an inspection package for a one- to three-line source code change outweighs the benefit. Second, the risk of delaying the inspection has proved to be acceptable. This risk is mitigated by the team's experience that small changes are typically identified in the IDENT phase, where a peer has already concurred with the change.
- **Code and Code Review.** Source code changes are made, based on baseline versions of the OFP. The software engineer reviews their changes using a code review checklist. Code inspection is deferred to the INSP phase.
- **Compile.** The source code is compiled until all code compiles cleanly.
- **Unit Test.** Unit testing is then performed. If the fix for the PR fails, the software engineer continues to solve the problem, iterating through design, code, and unit test until a successful solution is achieved and any negative side effects have been eliminated. When the desired results are obtained, the SE is shown the unit test results as an additional check that the solution is correct and complete. This conforms to the maintenance implementation ISO/IEC activity described in 5.5.3.2 (b) of Standard 12207 [1]
- **INSP.** The inspection phase consolidates both design and code inspections, except in cases where complex solutions require that design and code inspections be conducted separately. The inspection package includes the inspection log, modified design and source code files, the PR, a document describing where changes were made and why, and a copy of the unit test plan. The addition of the unit test plan accords with the maintenance implementation ISO/IEC activity as described in 5.5.3.2 (a) of Standard 12207 [1]. The addition of a document describing where and why changes were made is a road map for the inspector. The team made this a mandatory requirement when it was found that this information dramatically reduced inspection times.
- **IT.** Integration test is performed using a developmental baseline in the lab environment (i.e., the actual hardware). The software engineer uses the PR test plan, and the SE may perform additional tests to verify the correctness and completeness of the fix. If either the software engineer or the SE determines that the fix is incomplete, the software engineer continues to log his work in this phase. Once the software engineer and SE agree that the fix is correct, the PR is then considered completed and available for system testing; although, there are exceptions as indicated under the RA [rework assessment] and ST [system test] phases.
- **RA.** The rework assessment phase is added to a PR, in the TSP workbook, when questions arise about a completed PR that requires the software engineer to investigate and resolve. The software team found that there was often considerable time lag between completion of the IT phase and the closing of the PR by the SE. The SE would often have forgotten his or her initial consultation with the software engineer or would question whether the fix addressed something that may have been overlooked by the requirements. These questions would cause the software engineers to spend a considerable amount of time becoming reacquainted with a PR completed back under IT. This phase is used to capture time spent in determining if a software problem still exists for the PR in question. If no problem exists, the PR is closed by the SE. If a problem is identified, the ST phase is then entered.
- **ST.** When a problem is found in system test that is related to an allocated PR, an ST phase is added for that PR. The software engineer will log his or her time in this phase until the problem is resolved. Problems detected during the system test phase are used as the quality indicator for the H2.0 team. The team set its quality goal at having no more than 10% of PR's being rejected in system test.

IDENT	Identification
INWRK	In Work
INSP	Inspection
IT	Integration Test
RA	Rework Assessment
ST	System Test

Figure 2: *TSP Maintenance Life Cycle Phases*

A mapping of the development classic life-cycle phases to the maintenance lite life-cycle phases is shown in Figure 3 (see next page).

Customer's Perspective on TSP Maintenance Activities

The direct customer for the software team is the Block lead and the Integrated Product Team (IPT) lead. The customer defines project goals for maintenance projects and levels of success are established during TSP Meeting 1, just as in development projects. The customer participates in launches and postmortems,

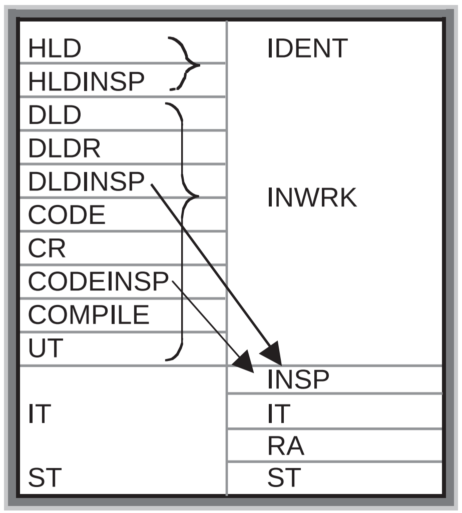


Figure 3: *Mapping from Development to Maintenance Life Cycle*

just as in development projects.

"TSP has brought credibility to estimates and commitments to perform worthwhile in a maintenance environment," said AV-8B JSSA's IPT Lead Dwayne Heinsma. "We are no longer questioned on the basis of estimating our requirements because we have the data and the performance to back it up."

Heinsma continued, "TSP has also contributed significantly to our ability to establish organizational improvement goals in the maintenance area. We did not have the historical measures that baselined our performance previously and today we now have a process by which we update our organizational performance baseline using project postmortem data."

Heinsma also stated, "Today, there is a strong push from NAVAIR leadership to establish improvement goals (productivity, quality, cost, and schedule) and show progress toward meeting those goals. With TSP, we have established the baseline; we have the improvement goals and the data showing progress toward the goals. If we are not meeting the goals, TSP provides us insight into what is holding us back, and we can focus on those elements that will help us improve most significantly."

The H4.0 Block Lead Greg Janson for the current software maintenance effort added, "I feel that TSP is worthwhile from a customer point of view. TSP provides a good quantitative tool for performance assessment. Qualitatively, it creates a solid basis for estimating that is difficult to question."

Results

The H2.0 team finished their tasking well ahead of schedule. In fact, the H2.0 team was able to reassess how many additional PR's could be solved in the time remaining on the project. The team initially estimated 102 PR's over a two-year period. The team actually completed over 180 PR's. This is a considerable accomplishment, given the team's relative lack of domain knowledge and the fact that the first two and one-half months of the project were spent completing TSP training. During this time, the H2.0 team also developed and documented processes for the H2.0 software development effort. The team came very close to their quality goal, achieving a 13 percent rejection rate in system test.

The AV-8B software team is now working on the next block upgrade to the AV-8B MSC OFP. They continue to refine the lite life cycle to improve software quality. In this block upgrade, their goal is to reduce the rework time measured in the RA phase, and to refine their defect logging to yield more fine-grained information earlier in the life cycle.◆

Reference

1. ISO [International Organization for Standardization]/IEC [International Electrotechnical Commission] 12207. ISO/IEC 12207: Information Technology – Software Life-Cycle Processes. 1st ed. ISO/IEC, 1995.

Notes

1. At the AV-8B JSSA, the systems engineering team is responsible for system and software requirements.
2. Common OFP is a basic set of software built upon for different aircraft with common fundamental requirements but differing missions and/or systems.

About the Author

Chris A. Rickets is a computer scientist in the software engineering group at the AV-8B Joint System Support Activity. He has been working on Embedded Avionics Systems for the past 14 years. He was the H2.0 Harrier Block Upgrade Software Lead and is currently working on the H4.0 Harrier Block Upgrade. Rickets has both a Bachelor of Science and Master of Science in computer science from California State University Chico.

CMDR, NAWCWD
41K300D MS 2004
507 E Corsair ST
China Lake, CA 93555-6110
Phone: (760) 939-5838
E-mail: chris.rickets@navy.mil

WEB SITES

Software Process Dashboard Initiative

http://processdash.sourceforge.net

The Software Process Dashboard Project is an open-source initiative to create a Personal Software Process℠ (PSP℠)/Team Software Process℠ (TSP℠) support tool. The Process Dashboard is an existing support tool originally developed in 1998 by the U.S. Air Force, and has continued to evolve under the open-source model. It is freely available for download under the conditions of the Gnu's Not Unix Public License. The Process Dashboard supports data collection, planning, tracking, data analysis, and data export. The major strengths of the Process Dashboard are ease of use, flexibility/extensibility, platform independence, and price. The Team Process Dashboard is currently under development.

Software Engineering Institute

www.sei.cmu.edu

The Software Engineering Institute (SEI℠) is a federally funded research and development center sponsored by the Department of Defense to provide leadership in advancing the state of the practice of software engineering to improve the quality of systems that depend on software. SEI helps organizations and individuals improve their software engineering management practices. The site features complete information on models the SEI is currently involved in developing, expanding, or maintaining, including the Team Software Process℠, Personal Software Process℠, Capability Maturity Model® Integration, Capability Maturity Model® for Software, Software Acquisition Capability Maturity Model®, Systems Engineering Capability Maturity Model®, and more.

Why Big Software Projects Fail: The 12 Key Questions

Watts S. Humphrey
The Software Engineering Institute

In spite of the improvements in software project management over the last several years, software projects still fail distressingly often, and the largest projects fail most often. This article explores the reasons for these failures and reviews the questions to consider in improving your organization's performance with large-scale software projects. Not surprisingly, considering these same questions will help you improve almost any large or small project with substantial software content. The principal questions concern why large software projects are hard to manage, the kinds of management systems needed, and the actions required to implement such systems. In closing, the author cites the experiences of projects that have used the methods described and cites sources for further information on introducing the required practices.

Software project failures are common, and the biggest projects fail most often. There are always many excuses for these failures, but there are a few common symptoms. Some years ago, before the invention of the Capability Maturity Model® (CMM®) and CMM Integration℠ (CMMI®) the principal problem was the lack of plans [1, 2]. In the early years, I never saw a failed project that had a plan, and very few unplanned projects were successful.

The methods defined for CMM and CMMI Levels 2 and 3 helped to address this problem. As the Standish data in Figure 1 shows, the success rate for software organizations improved between 1994 and 2000, and much of this improvement was due to more widespread use of sound project management practices [3]. Still, with less than 30 percent of our projects successful, those of us who are software professionals have little to be proud of.

The definition of a successful project is one that completed within 10 percent or so of its committed cost and schedule and delivered all of its intended functions. Challenged projects are ones that were seriously late or over costs or had reduced functions. Failed projects never delivered anything. Figure 2 (see page 26) shows another cut of the Standish data by project size. When looked at this way, half of the smallest projects succeeded, while none of the largest projects did. Since large projects still do not succeed even with all of the project management improvements of the last several years, one begins to wonder if large-scale software projects are inherently unmanageable.

Question 1: Are All Large Software Projects Unmanageable?

There are some large, unprecedented projects that are so risky that they would likely be challenged under almost any management system. But some large projects have succeeded. Two examples are the Command Center Processing and Display System Replacement (CCPDS-R) project, described by Walker Royce, and the operating system (OS)/360 project in my former group at IBM [4, 5]. The CCPDS-R was a U.S. Air Force installation at Cheyenne Mountain in Colorado. It had about 100 developers at its peak. The OS/360 was the operating system to support the IBM 360 line of computers, and included the control program, data management, languages, and support utilities. Its development team consisted of about 3,000 software professionals.

Both of these projects placed heavy emphasis on planning, and both adopted an evolutionary development strategy with multiple releases and phased specifications. Both projects also took a somewhat unconventional approach to motivating team member performance. For CCPDS-R, management distributed 50 percent of the project award fee to the development team members. This built their loyalty and commitment to success, and maintained team motivation throughout the job. The CCPDS-R project was delivered on schedule and within contracted costs.

By the time I took over the OS/360 project some years ago, we had all learned that the proper strategy for building big software-intensive systems was to break the job into as many small incremental releases as practical. Since this strategy required organization-wide coordination, our very first action was to have all the development teams in all the involved laboratories produce their own plans and coordinate them through a central build-and-release group. Then, we based the company's commitments on the dates that the teams provided. In no case did IBM commit to any date that was not supported by a plan that had been developed by the team that was to do the work.

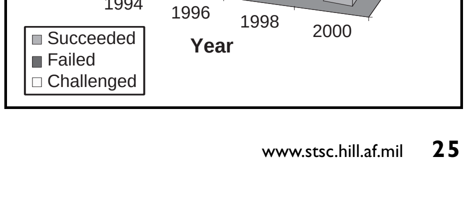
These plans extended through 19 releases over a period of 30 months. Most importantly, they provided the focus we all needed to coordinate the work of 15 laboratories in six countries and to promptly recognize and address the myriad problems that inevitably arose. The developers were personally committed to their schedules, and they delivered every one of these releases on or ahead of the committed schedules. So, at least based on this limited sample, some large software projects can be managed successfully. However, because the success rate is so low, large-scale software projects remain a major project management challenge.

Question 2: Why Are Large Software Projects Hard to Manage?

While large software projects are undoubtedly hard to manage, the key question is "Why?"

Historically, the first large-scale management systems were developed to manage armies. They were highly autocratic, with the leader giving orders and the

Figure 1: *Project Success History [3]*



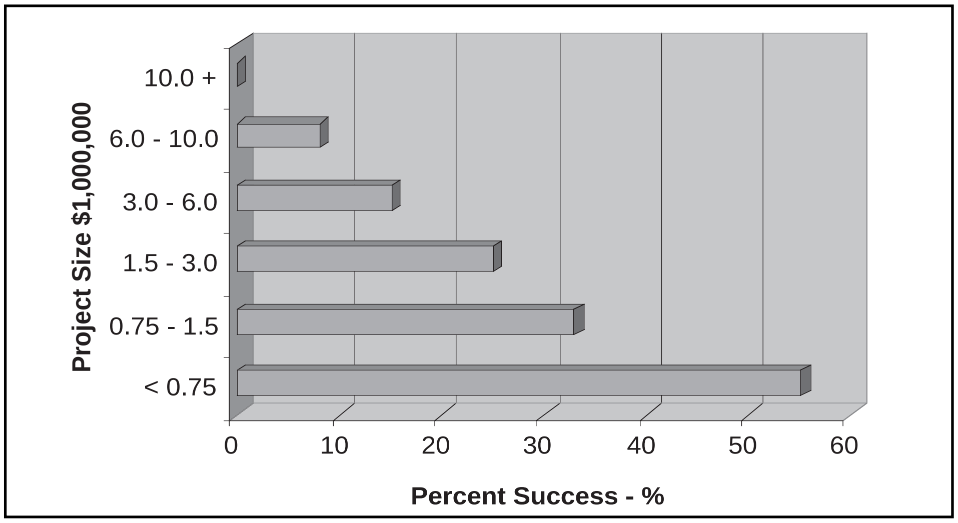


Figure 2: *Success Rate by Project Size [3]*

troops following. Over time, work groups were formed for major construction projects such as temples, palaces, fortifications, and roads. The laborers were mostly slaves, and again, the management system was highly autocratic. The workers did what they were told or they were punished.

This army-like structure was essentially the only management system for many years until the Greek city-states introduced democratic political systems. However, these democratic principles were primarily used for governing, not for project management. Somewhat later, a totally different management system was used to build cathedrals. This work was largely done by volunteer artisans who managed themselves under the guidance of a master builder. Since building a cathedral often took 50 years or more, the cathedral-management system is not a good model for modern large-scale software projects. However, it did produce some beautiful results. This cathedral-building management system was not used for anything but cathedrals for many years, but it has recently had some success as the guiding principle for the open source software development community [6].

The next major management innovation was the factory. Factories started producing clothing and were soon used for making all kinds of goods. Again, however, the factory management system was autocratic, with management directing and workers doing. While the factory model improved productivity, it was not without its problems. The early work of Frederick Winslow Taylor about 100 years ago and the more recent work of W.E. Deming, J.M. Juran, and others has improved the effectiveness of this model by redefining the role of the worker. The modern view is that to do quality work for predictable costs and schedules, workers must be treated as thinking and feeling participants rather than merely as unfeeling drudges. However, to date, these methods have had limited application to software [7, 8, 9].

The factory/army system has persisted and now characterizes the modern corporate structure where senior management decides and everybody else follows. Many managers would contend that they listen to their people while making decisions. However, employees generally view corporate management as autocratic and few feel that they could influence a senior manager's decisions. Some managers even argue that autocratic management is the only efficient style for running large projects and organizations. Democratic debates would take too long and decisions would not be made by the most important or knowledgeable people.

Regardless of the validity of this view, the hierarchical management style does not work well for managing large software projects. Unfortunately, except for the cathedral-building system, there is no other proven way to manage large-scale work. So, if we want to have successful large-scale software projects, we must develop a project management system that is designed for this purpose.

The first large-scale management systems were developed for armies. Leaders gave orders and troops followed. With training and discipline, this approach could work even amid chaos and confusion.

Large-scale management systems were eventually applied to major construction projects. The system was highly autocratic; workers did what they were told or they were punished.

Question 3: Why Is Autocratic Management Ineffective for Software?

Before developing a new management system, we should first understand why the current one does not work. To answer this question, we must explore the nature of software work and how it differs from other, more manageable work. Software and software-like work have characteristics that are particularly difficult to manage. From a management perspective, the principal difference between managing traditional hardware projects and modern software work concerns management visibility.

With manufacturing, armies, and traditional hardware development, the managers can walk through the shop, battlefield, or lab and see what everybody is doing. If someone is doing something wrong or otherwise being unproductive, the manager can tell by watching for a few minutes. However, with a team of software developers, you cannot tell what they are doing by merely watching. You must ask them or carefully examine what they have produced. It takes a pretty alert and knowledgeable manager to tell what software developers are doing. If you tell them to do something else or to adopt a new practice, you have no easy way to tell if they are actually working the way you told them to work.

Some might argue that hardware work is not actually that different from software work and that, at least for some hardware tasks and most system engineering jobs, the work is equally opaque to management. This is certainly true, particularly when the hardware engineers are producing microcode, using hardware design languages, or working with simulation or layout tools. Today, as modern technical specialties increasingly overlap, many hardware projects now share the same characteristics as large software projects. When hardware development and system-engineering work have the characteristics of software work, they should be managed like software. However, since these systems groups generally tend to be relatively small, they do not yet present the same project-manageability problems as large-scale software.

Question 4: Why Is Management Visibility a Problem for Software?

Since most software developers are dedicated and hard-working professionals, why is management visibility a problem?

The problem is that the manager cannot tell where the project stands. To manage modern large-scale technical work, you must know where the project stands, how rapidly the work is being done, and the quality of the products being produced. With earlier hardware-development projects, all of this information was more-or-less visible to the manager, while with modern software and systems projects it often is not.

This is a problem because large development projects, whether hardware or software, always run into problems, and every problem involves more work. While developers can invariably overcome small problems, every problem adds to the workload and delays the job. Each little slip is generally manageable by itself, but over time, problems add up, and sooner or later the project is in serious trouble.

The project manager's job is to identify these small daily slips and to take steps to counter them. As Fred Brooks said, "Projects slip a day at a time" [10]. With traditional hardware projects, the manager could usually see these one- and two-day slips and could do something about them. With modern, complex, software-intensive systems, the daily schedule slips are largely invisible. So, with large-scale software work, the managers generally do not see the schedule problem until it is so big that it is obvious. Then, however, it is usually too late to do much about it.

Question 5: Why Can't Managers Just Ask the Developers?

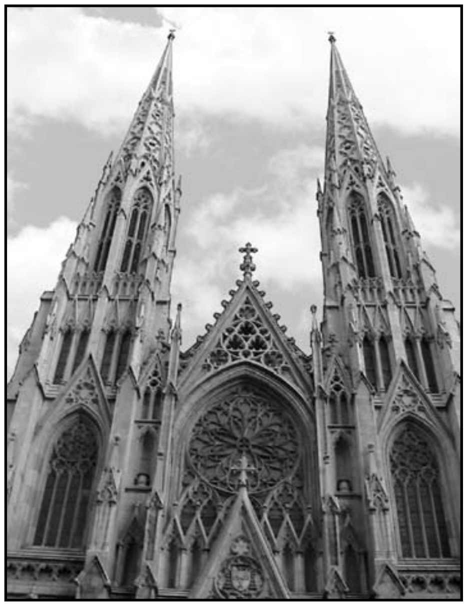
If the managers cannot see where the developers stand, why not just ask them?

Most developers would be glad to tell their managers where they stood on the job. The problem is that, with current software practices, the developers do not know where they stand any more than the managers do. The developers know what they are doing, but they do not have personal plans, they do not measure their work, and they do not track their progress. Without these practices to guide them, software people do not know with any precision where they are in the job. They could tell the manager that they are pretty close to schedule or 90 percent done with coding, but the fact is that they do not really know. Again, as Brooks said, "...programmers generally think that they are 90 percent through with the coding for more than half of the project" [10].

Unless developers plan and track their personal work, that work will be unpredictable. Furthermore, if the cost and schedule of the developers' personal work is unpredictable, the cost and schedule of their teams' work will also be unpredictable. And, of course, when a project team's work is unpredictable, the entire project is unpredictable. In short, as long as individual developers do not plan and track their personal work, their projects will be uncontrollable and unmanageable.

Anyone who has managed software development will likely argue that this is an overstatement. Although you may not know precisely where each developer's work stands, you can usually get a general idea. Since about a third to a half of the small projects are successful when the developers do not plan and track their personal work, such projects can be managed. So why should the lack of sound personal software practices be a problem for large projects?

It is true that software projects are not totally unmanageable. As Figures 1 and 2 show, the worst problem is with the very large software projects. On small projects, some uncertainty about each team member's status is tolerable. However, as projects get bigger and communications lines extend, precise status information becomes more important. Without hard data on project status, people communicate opinions, and their opinions can be biased or even wrong. When filtered through just a few layers of management, imprecise project status reports become so garbled that they provide little or no useful information. Then these large-scale software projects end up being run with essentially no management visibility into their true status, issues, and problems.


Work on cathedrals was done by volunteer artisans who managed themselves under the guidance of a master builder. This approach has had some success in open source software development.

Question 6: Why Do Planned Projects Fail?

Today, with CMM and CMMI, most large software projects are planned, and they use methods like Program and Evaluation and Review Technique (PERT) and earned value to track progress. Why is that not adequate?

The problem is with the imprecision and inaccuracy of most software project plans. Most projects have major milestones such as specifications complete, design complete, code complete, and the like. The problem is that on real software projects, few of these high-level tasks have crisp completion dates. The requirements work generally continues throughout design and even into implementation and test; coding usually starts well before design completion and continues through most of testing.

A few years ago, the management of a large software organization asked me to review their largest project. They told me that the code completion milestone had already been met on schedule. However, I found that very little code had actually been released to test. When I met with the development teams, they did not know how much code they had written or what remained to be done. It took a full week to get a preliminary count, and it was a month before we got accurate data. It was another 10 months before all of the coding was actually completed. It is not that developers lie, just that without objective data, they have no way to know precisely where they stand. When they are under heavy schedule pressure, people try to respond. Since we all know that the bearer of bad news tends to be blamed, no one dares to question the schedule and everyone gives the most optimistic story they can.

Question 7: Why Not Just Insist on Detailed Plans?

Why cannot management just insist on more detailed plans? Then they could have more precise measures of project status.

While this would seem reasonable, the issue is, "Whose plans are they?" Detailed plans define precisely how the work is to be done. When the managers make the plans, we have the modern-day equivalent of laborers building pyramids. The managers tell the workers what to do and how to do it, and the workers presumably do as they are told.

While this has been the traditional approach for managing labor, it has become progressively less effective for managing high-technology work, particularly software. The principal reason is that the managers do not know enough about the work to make detailed plans. That is why many of these software-intensive projects typically have very generalized plans. This provides the developers with the flexibility they need to do creative work in the way that they want to. The current system is therefore the modern equivalent of the cathedral-building system where the developers act like artisans. The unfortunate consequence is that, without Herculean effort, it often seems that the natural schedule for such projects could easily approach 50 years.

Question 8: Why Not Tell the Developers to Plan Their Work?

The obvious next step would be to tell the developers to make their own detailed plans. Why would this not work?

There are three problems. First, most developers do not want to make plans; they would rather write programs. They view planning as a management responsibility. Second, if you told them to make plans, they would not know how to do it. Few of them have the skill and experience to make accurate or complete plans. Finally, making accurate, complete, and detailed plans means that the developers must be empowered to define their own processes, methods, and schedules. Few managers today would be willing to cede these responsibilities to the software developers, at least not until they had evidence that the developers could produce acceptable results.

Question 9: How Can We Get Developers to Make Good Plans?

It seems that the problem of effectively managing large software projects boils down to two questions: How can we get the software developers and their teams to properly make and faithfully follow detailed plans, and how can we convince management to trust the developers to plan, track, and manage their own work?

To get the developers to make and follow sound personal plans, you must do three things: provide them with the skills to make accurate plans, convince them to make these plans, and support and guide them while they do it.

Providing the skills is just a question of training. However, once the developers have learned how to make accurate plans and to measure and track their work against these plans, they usually see the benefits of planning and are motivated to plan and track their own and their team's work. So, it is possible that developers *can* be taught to plan and, once they learn how, they are generally willing to make and follow plans [11].

Question 10: How Can Management Trust Developers to Make Plans?

This is the biggest risk of all: Can you trust developers to produce their own plans and to strive for schedules that will meet your objectives?

This question gets to the root of the problem with autocratic management methods: trust. If you trust and empower your software and other high-technology professionals to manage themselves, they will do extraordinary work. However, it cannot be blind trust. You must ensure that they know how to manage their own work, and you must monitor their work to ensure that they do it properly. The proper monitoring attitude is not to be distrustful, but instead, to show interest in their work. If you do not trust your people, you will not get their whole-hearted effort and you will not capitalize on the enormous creative potential of cohesive and motivated teamwork. It takes a leap of faith to trust your people, but the results are worth the risk.

Question 11: What Are the Risks of Changing?

Every change involves some risk. However, there is also a cost for doing nothing. If you are happy with how your large software projects are performing, there is no need to change. However, few managers or professionals are comfortable with the current state of software practice, particularly for large-scale projects. So, there are risks to changing and risks to not changing. The management challenge is to balance these risks before deciding what to do.

There are two risks to changing to a new management system for large-scale software projects. First, it costs time and money to train the developers to plan and track their work and to train the managers to use a new management system. Then comes the risk of using these methods on a real project. While you will see some early benefits, you will not know for sure whether this new management system is truly effective for you until the first project is completed and you can analyze the results.

This brings up a related and even more difficult problem: On large multi-year projects, there is not time to run pilots. You must pick a management strategy and go with it. However, since almost all large software-intensive projects are now failing anyway, the biggest risk is *not changing*. Perhaps the biggest shock for most managers is realizing that they are part of the problem, and that they have to change their behavior to get the kind of large-system results they want.

These problems are common to all change efforts. The way to manage these problems is to examine the experiences of others and to minimize your exposure by carefully planning your change effort and getting help from people who have already used the methods you plan to introduce. Of course the alternative is to hope that things will get better without any changes. With this choice, however, your large-systems projects will almost certainly continue to perform much as they have in the past.

Question 12: What Has Been the Experience So Far?

The Software Engineering Institute (SEI^{SM}) has developed a method called the Team Software Process^{SM} (TSP^{SM}) that follows the concepts described in this article [11]. With the TSP[1], if you properly train and support your development people and if you follow the SEI's TSP introduction strategy, your teams will be motivated to do the job properly. The team members' personal practices will be defined, measured, and managed; team performance will also be defined, measured, and managed; and the project's status and progress will be precisely reported every week. Although this will not guarantee a successful project, these practices have worked for the several dozen projects that have tried them so far.

Moreover, there is one caveat. These

practices have proven effective for teams of up to about 100 members, as well as for teams composed of multiple hardware, systems, and software professionals. They have even worked for distributed teams from multiple geographic locations and organizations. Although these methods should scale up to very large projects, the TSP has not yet been tried with projects of over 100 professionals. I know from personal experience, however, that these practices will address many of the problems faced by the managers of software organizations of several thousand developers.

The other articles in this issue describe the TSP experiences of several organizations. They describe how these practices have worked on various kinds of projects and how they could help your organization.◆

Acknowledgements

Many people have participated in the work that led to this article, so I cannot thank them all personally. However, without their willingness to try new methods and to take the risks that always accompany change, this work would not have been possible. So, to everyone who participated in the early CMM and CMMI work and to all of those who have learned and used the Personal Software Process^{SM} and TSP, you have my profound gratitude. I have also had the advice and support of several people in writing this article. My special thanks go to Dan Burton, Noopur Davis, Bill Peterson, Marsha Pomeroy-Huff, and Walker Royce.

References

1. Humphrey, Watts S. Managing the Software Process. Reading, MA: Addison-Wesley, 1989.
2. Chrissis, Mary Beth, Mike Konrad, and Sandy Shrum. CMMI – Guidelines for Process Integration and Process Improvement. Reading, MA: Addison Wesley, 2003.
3. The Standish Group International, Inc. Extreme Chaos. The Standish Group International, Inc., 2001.
4. Royce, Walker. Software Project Management, A Unified Framework. Reading, MA: Addison-Wesley, 1998.
5. Humphrey, Watts S. "Reflections on a Software Life." In the Beginning, Recollections of Software Pioneers. Robert L. Glass, Ed. Los Alamitos, CA: IEEE Computer Society Press, 1998.
6. Raymond, Eric S. The Cathedral and the Bazaar. Cambridge, MA: O'Reilly Publishers, 1999.
7. Deming, W. Edwards. The New Economics for Industry, Government, Education. 2nd ed. The MIT Press, Cambridge, MA, 2000.
8. Juran, J.M., and Frank M. Gryna. Juran's Quality Control Handbook, Fourth Edition. New York: McGraw-Hill Book Company, 1988.
9. Taylor, Frederick Winslow. The Principles of Scientific Management. New York: Harper and Row, Publishers, Inc., 1911.
10. Frederick P. Brooks. The Mythical Man-Month. Reading, MA: Addison Wesley, 1995.
11. Humphrey, Watts S. Winning With Software: An Executive Strategy. Reading, MA: Addison-Wesley, 2002.

Note

1. The Software Engineering Institute offers courses and transition services to help organizations introduce the TSP. Additional information is available at <tsp@sei.cmu.edu> or at <www.sei.cmu.edu/tsp>.

About the Author

Watts S. Humphrey joined the Software Engineering Institute (SEI^{SM}) of Carnegie Mellon University after his retirement from IBM in 1986. He established the SEI's Process Program and led development of the Software Capability Maturity Model®, the Personal Software Process^{SM}, and the Team Software Process^{SM}. During his 27 years with IBM, he managed all IBM's commercial software development and was vice president of Technical Development. He holds graduate degrees in physics and business administration. He is an SEI Fellow, an Association for Computing Machinery member, an Institute of Electrical and Electronics Engineers Fellow, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. He has published several books and articles and holds five patents.

Carnegie Mellon University
4500 Fifth AVE
Pittsburgh, PA 15213-2612
Phone: (941) 924-4169
Fax: (941) 925-1573
E-mail: watts@sei.cmu.edu

The Joint Services

SSTC

Systems & Software Technology Conference

18 - 21 April 2005 • Salt Lake City, UT

“Capabilities: Building, Protecting, Deploying”

SSTC is the premier forum in the Department of Defense (DoD) to enhance attendees   professional skills and knowledge of systems and software technologies and policies; enabling them to improve the capabilities they provide to the warfighter

Design your own stimulating conference experience by choosing from the following sessions and by attending the cutting-edge trade show:

<u>Sponsored/Special Sessions</u>	<u>Track Topics</u>
Government Co-Sponsor Panel Disussion	Security
Industry Panel Discussion	Software
SEI Acquisition	Acquisition
OSD	Systems
INCOSE	Management/Policy
STSC	Process Improvement
CIO	Innovations
Microsoft	
DISA	
IEEE	
Plenary Sessions	
Hands-On Computer Lab	
Poster Sessions	

Visit our web site or call

www.stc-online.org

800-538-2663

Conference & Exhibit Registration

Now Open - Register Today!

The Joint Services Systems & Software Technology Conference is co-sponsored by:

United States Army

United States Marine Corps

United States Navy

Department of Navy

United States Air Force

Defense Information Systems Agency

A Few Good Launch Coaches

Welcome to "Backfire," the interview within a journal where we cross-examine popular icons for software truth. This month we have Col. Nathan R. Jessep, former commanding officer, Marine Ground Forces, Guantanamo Bay, Cuba, from the movie "A Few Good Men." After a dishonorable discharge, Mr. Jessep has started a new career as a Team Software Process^{SM} (TSP^{SM}) Launch Coach.

Petersen: Welcome Mr. Jessep.
Jessep: Colonel.

Petersen: What's that?
Jessep: I would appreciate it if you would address me as Colonel or Sir. I believe I have earned it.

Petersen: OK. Colonel, let us focus on your new career. Why did you implement Personal Software Process^{SM} (PSP^{SM}) and TSP on your project?
Jessep: I felt that our agile developers might be in danger without disciplined measurement.

Petersen: Grave danger?
Jessep: Is there another kind?

Petersen: Is it true you implemented TSP to drive out agile developers?
Jessep: No, I specifically gave orders for project managers to coddle developers with agile tendencies.

Petersen: Any chance they ignored that order?
Jessep: Ignored the order?

Petersen: Any chance they forgot about it?
Jessep: No.

Petersen: Any chance they left your office and thought, "The old man is wrong?"
Jessep: Have you ever spent time in a TSP unit, son?

Petersen: No, sir.

Jessep: Ever calculated earned value?
Petersen: No, sir.

Jessep: Ever put your project in another man's hands; ask him to put his project in yours?
Petersen: No, sir.

Jessep: We follow processes, son. We follow processes or projects die. It is that simple. Are we clear?
Petersen: Yes, sir.

Jessep: Are we clear!
Petersen: Crystal.

Petersen: Colonel, if you have sound processes and developers always follow those processes, why did you order managers to coddle agile programmers? Why the extra order?
Jessep: The agile developers were substandard programmers –

Petersen: But that is not what you said. You said your agile developers might be in danger without disciplined measurement. I said, 'Grave danger.' You said –
Jessep: I know what I said.

Petersen: Then why the order sir?
Jessep: Sometimes men take matters into their own hands.

Petersen: No, sir. You made it clear that your men never take matters into their own hands. Your men follow processes or projects die. Therefore, the project should not have been in any danger, Colonel.
Jessep: You pretentious wimp. You want answers?

Petersen: I think I am entitled to them.
Jessep: You want answers!

Petersen: I want the truth.
Jessep: You can't handle the truth!

Jessep: Son, we live in a world run by software and that software has to be developed by men with discipline. Who is going to do it? You, Petersen? Bill Gates? I have a greater responsibility than you can possibly fathom. You weep for agile developers and you curse PSP. You have that luxury. You have the luxury of not knowing what I know: That the agile developers' dismissal, while tragic, probably saved that project and my existence, while grotesque and incomprehensible to you, saves projects.

You don't want the truth. Because deep down, in places you don't talk about at parties, you want me developing software. You need my software. We use words like process, measurement, maturity … we use these words as the backbone of a life spent developing software. You use them as a punch line.

I have neither the time nor the inclination to explain myself to a man who rises and sleeps under the blanket of the very software I provide, and then questions the manner in which I provide it. I would prefer you just said thank you and went on your way. Otherwise, I suggest you pick up a keyboard and code a module. Either way, I don't give a darn what you think you are entitled to.

Petersen: Did you order their dismissal?
Jessep: I did my job and I would do it again.

Petersen: Did you order their dismissal!
Jessep: You're darn right I did!

OK, thank you, Colonel Jessep. Join us next time when "Backfire" cross-examines the cast of Seinfeld on capability maturity models.

— Gary A. Petersen
Shim Enterprise, Inc.

Can You BackTalk?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CrossTalk, we also accept articles for the BackTalk column. BackTalk articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BackTalk article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BackTalk article, visit our Web site at <www.stsc.hill.af.mil>.

TSP℠ ENGINEERS in a CMM® LEVEL 5 ORGANIZATION:
a WINNING COMBINATION
Team Software Process℠ is helping the MAS Software Engineering Division at Ogden Air Logistics Center hit home runs.
WINNING PROJECTS and TSP TEAMS at MAS
TASKVIEW
AIR TASKING ORDER PARSER
GROUND THEATER AIR CONTROL SYSTEM
GTACS
GROUND THEATER AIR CONTROL SYSTEM
GTACS
SYSTEM PROGRAM OFFICE
MAS
IS RUNNING THE BASES WITH...
PRODUCTIVITY IMPROVEMENTS
ACCURATE SCHEDULE AND COST ESTIMATES
LOW DEFECTS
DISCIPLINED ENGINEERING TEAMS
HIGH CUSTOMER SATISFACTION
LOW COST SOLUTIONS and HIGH QUALITY PRODUCTS

Contact MAS Today:
Software Engineering Division
7278 4th Street
Ogden Air Logistics Center
Hill Air Force Base, Utah
84056-5205

DSN: 777-2615
Commercial: (801) 777-2615
E-mail: OOALC.ContactMAS@hill.af.mil
Or visit our website: www.mas.hill.af.mil

MAS
Software Engineering Division
Ogden Air Logistics Center

Co-Sponsored by
U.S. Air Force
Air Logistics Centers
MAS Software Divisions

CrossTalk / MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737